SB21-3015-0

Installed User Program

Language Manual

APL2

Program Number: 5798-DJP

A programming language (APL) is a general purpose language which is used in a variety of applications, such as commercial data processing, systems design, prototyping, and scientific and engineering computation. It has proven particularly useful in data manipulation applications, where its computational power and interactive facilities combine to enhance the productivity of both application programmers and end users.

APL2 provides major enhancements over previous IBM APL implementations. Extensions and improvements have been made to the language and environmental facilities as well as to the internal structure and operation of the system.

This manual describes the APL2 programming language and provides a reference for users.

IBM

PROGRAM SERVICES

Central Service will be provided until otherwise notified. Users will be given a minimum of six months notice prior to the discontinuance of Central Service.

During the Central Service period, IBM through the program sponsor(s) will, without additional charge, respond to an error in the current unaltered release of the program by issuing known error correction information to the customer reporting the problem and/or issuing corrected code or notice of availability of corrected code. However, IBM does not guarantee service results or represent or warrant that all errors will be corrected.

Any on-site program service or assistance will be provided at a charge.

WARRANTY

EACH LICENSED PROGRAM IS DISTRIBUTED ON AN 'AS IS' BASIS WITHOUT WARRANTY OF ANY KIND EITHER EXPRESS OR IMPLIED.

Central Service Location: IBM Corporation
APL Development/Department J88
555 Bailey Avenue (P.O. Box 50020)
San Jose, CA 95150
Telephone: (408) 463-APL2
Tieline: 8-543-APL2

First edition (June 1982)

This is the initial edition of the APL2 Language Manual (SB21-3015). This edition applies to Release 1 of the APL2 Installed User Program, product number 5798-DJP, and to any subsequent releases until otherwise indicated in new editions or technical newsletters.

It is possible that this material may contain reference to, or information about, IBM products (machines and programs), programming, or services that are not available in your country. Such references or information must not be construed to mean that IBM intends to announce such products in your country.

Publications are not stocked at the address given below; requests for IBM publications should be made to your IBM representative or to the IBM branch office serving your locality.

A form for reader's comments is provided at the back of this publication. If the form has been removed, comments may be sent to the Central Service Location address listed above.

IBM is also interested in any comments you might have, either good or bad, on the new features provided by APL2 and the CMS implementation of it. Suggestions you provide will be considered for possible inclusion in future releases of APL2 products.

IBM accepts such comments with the understanding that it may use or distribute whatever information you supply in any way IBM believes appropriate without incurring any obligation to you.

PREFACE

This manual describes the APL2 programming language. An introduction is given in the first two chapters. APL2 contains extensions to APL which include computation on complex numbers, computation on nested arrays, and computation on functions (with operators).

The Contents, Index, and certain Figures, are significant parts of this manual. The Contents gives the organization of the manual and the general structure of APL2 itself. The Index includes entries for each language entity by both name and purpose, as well as for language attributes, system commands and messages, and miscellaneous items. Notable figures are:

1. Primitive Pervasive Functions (Figure 2 on page 29)

2. Primitive Non-Pervasive Functions (Figure 3 on page 32)

3. Primitive Operators (Figure 8 on page 155)

4. System Functions (Figure 11 on page 181)

5. System Variables (Figure 12 on page 203)

6. System Labels (Figure 13 on page 227)

7. The APL2 Character Set (Figure 17 on page 285)

The first six figures are lists of various language items. Provided with each item is a page reference where the full description of that item can be found.

Throughout this manual, examples of APL2 statements make frequent use of redundant blanks for emphasis. The pair of symbols ←→ is used to indicate "is equivalent to".

LIST OF ILLUSTRATIONS

The characteristics of APL2 can be summarized as follows:

1. There are only three kinds of objects.

   a. Variables are named arrays which contain data.

   b. Functions act on arrays.

   c. Operators act on functions.

2. The syntax is simple.

   a. Operators have higher precedence than functions.

   b. There is no precedence hierarchy among operators.

   c. There is no precedence hierarchy among functions.

   d. Defined functions and operators (which may be called programs) are treated like primitive functions and operators.

3. The semantic rules are few.

   a. The definitions of primitive functions are independent of the representations of the data to which they apply.

   b. All pervasive functions apply to arrays in the same way.

   c. Primitive functions and operators have no hidden side effects.

4. The sequence control in a program is simple.

   a. One statement type embraces all types of branches (such as conditional, unconditional, and computed).

   b. The termination of the execution of any function or operator always returns control to the point of use.

5. External communication is established by variables which are shared between APL2 programs and other systems, subsystems, the APL2 environment, or auxiliary processors. These shared variables are treated both syntactically and semantically like other variables.

INTERACTION

APL2 is implemented as an interactive computer language. This means that you and the computer take part in a dialog. Your part of the dialog is normally indented 6 spaces, and the computer's part of the dialog normally begins at the left margin. This is called immediate execution, or calculator mode. Computer processing or the display of output can be interrupted by a terminal attention.

Lines being entered on non-display terminals can be corrected before entry by backspacing to the point of error, entering an attention, and re-entering from that point.

Programs of APL2 statements can be written and stored for subsequent automatic execution.

WORKSPACES

The common organizational unit of the APL2 system is the _workspace_. When in use, a workspace is said to be active, and it occupies a block of main storage. Part of each workspace is set aside to serve the internal workings of the system, and the remainder is used, as required, for storing programs and items of (transient and permanent) information.

Inactive workspaces are stored in libraries, which occupy space in auxiliary storage. Copies of inactive workspaces can be made active, or selected information can be copied from them into an active workspace.

ARRAYS

The basic unit of information in APL2 is an _array_. An array is an ordered rectangular collection of data _elements_. The number of dimensions of an array is called its _rank_. The collection of the lengths of all the dimensions of an array is called its _shape_.

An array of rank 0 is called a _scalar_.

An array of rank 1 is called a _vector_.

An array of rank 2 is called a _matrix_.

An array of rank 2 or greater is called a multi-dimensional array.

A dimension is also called an axis. Individual elements of an array are scalars. A scalar element may be a number, a character, or an enclosed array of arbitrary rank. The distinction between a number and a character is called type. An array which has only scalar numbers, or characters, or both as its elements is called a simple array. An array which has both scalar numbers and characters as its elements is called a mixed array.

An array which has at least one element which is not a number or a character is called a non-simple array or a nested array. A disclosed element of an array is called an item of the array. An enclosed item of an array is called an element of the array. An item is the array data within the structure of an element.

An array which contains at least one 0 in its shape is called an empty array. An empty array has no elements, but it has type, and possibly nested structure. An empty array may be either simple or non-simple.

NUMBERS

All numbers are entered or displayed in decimal. They may be either in conventional form (including a decimal point if appropriate), or in scaled form.

The scaled form consists of three consecutive parts:

1. an integer or decimal fraction called the mantissa, or multiplier

2. the letter *E*

3. an integer called the scale (which must not include a decimal point)

The scale specifies the power of 10 by which the mantissa is multiplied.

Examples:

```
      12E0
12

      12E3
12000
```

```
      12.34E3
12340

      12.3456E3
12345.6
```

Negative numbers are represented by an overbar (¯) immediately preceding the number. In scaled form, the multiplier and the scale may both be negative.

Examples:

```
      ¯12E3
¯12000

      12E¯3
0.012

      ¯12E¯3
¯0.012
```

The overbar (¯) used to start a negative numeric constant is distinguished from the bar (-) which denotes the negation function.

CHARACTERS

Characters are entered within a pair of quotes. The surrounding quotes are not displayed on output.

Examples:

```
      'C'
C

      '*'
*
```

The quote character itself must be entered as a pair of quotes in a character constant.

Example:

```
      ''''
'
```

NAMES

APL2 objects (variables, functions, and operators) may be given names. An object name may be any sequence of alphanumeric characters which starts with an alphabetic character. Alphabetic and alphanumeric characters are listed in "The APL2 Character Set" on page 285. A name may not contain a blank.

SPECIFICATION OF VARIABLES

An undefined name or the name of a variable may be assigned an array value by specification with the left arrow (←).

Examples:

```
A ← 1

B4 ← 2

AN¯INTEGER ← 4

A_¯_NUMBER ← ¯4

COST ← 5.98

∆X ← 0.1

PART⍙NUMBER ← 606

CHARACTER ← '/'
```

There may be multiple specifications in the same expression:

```
X ← 1 + Y ← 1
```

The result of a specification is the value being specified, so that *Y* is given the value 1, and *X* is given the value 2.

VECTOR NOTATION

The juxtaposition of two or more arrays in an expression results in a vector whose items are the arrays. This is called vector notation.

Example:

```
      A ← 2 4 8
```

The shape of vector *A* is 3:

```
      ⍴A
3
```

Its elements are scalars.
Its first element is 2:

```
      A[1]
2
```

Its second element is 4:

```
      A[2]
4
```

Its third element is 8:

```
      A[3]
8
```

Numbers and characters may be in the same vector.

Example:

```
      A ← 2 'X' 8

      ⍴A
3

      A[1]
2
      A[2]
X
      A[3]
8
```

Unless a character item is adjacent to a numeric item or a name, either blanks or parentheses must separate the items in vector notation.

```
      1 2   ↔   1 2
      1 2   ↔   1(2)
      1 2   ↔   (1)2
      1 2   ↔   (1)(2)

      2 'X' 8   ↔   2'X'8
      2 'X' 8   ↔   2('X')8
```

Throughout this manual, the pair of symbols ↔ is used to indicate "is equivalent to".

More than one consecutive blank or set of parentheses is redundant, but permitted.

```
        1 2  ↔  1  2
        1 2  ↔  1 (2)
        1 2  ↔  (1) 2
        1 2  ↔  (1) (2)

        1 2  ↔  1   2
        1 2  ↔  1((2))
        1 2  ↔  ((1))2
        1 2  ↔  ((1))((2))
```

Characters in a vector consisting of only characters may be listed between a single set of quotes:

```
        'ABC'  ↔  'A' 'B' 'C'
        'ABC'  ↔  'A'  'B'  'C'
```

The quote character itself must be entered as a pair of quotes in a character vector constant.

Example:

```
      A ← 'DON''T'
      A
DON'T
      ρA
5
```

Blanks within quotes are significant.

Example:

```
      A ← 'DO NOT'
      A
DO NOT
      ρA
6
```

Character vectors may themselves be items in vector notation.

Example:

```
      A ← 2 'MORE' 'TIMES'
```

The shape of vector *A* is 3:

```
      ρA
3
```

Its first element is 2:

```
      A[1]
2
```

The item contained in its second element is '*MORE*':

```
      ⊃A[2]
MORE
```

The item contained in its third element is '*TIMES*':

```
      ⊃A[3]
TIMES
```

Parentheses may be used to delimit items in vector notation.

Example:

```
      A ← 1 (2 3 4 5)
      ρA
2
      A[1]
1
      ρA[2]

      ⊃A[2]
2 3 4 5
      ρ⊃A[2]
4
```

If parentheses are used to separate items, then blanks are not required, but permitted. If parentheses are used to identify items, then computation may be done inside the parentheses.

Example:

```
      A ← 1 (2 3ρι6)
      ρA
2
      A[1]
1
      ⊃A[2]
1 2 3
4 5 6
      ρ⊃A[2]
2 3
```

Vector notation may be nested.

Example:

```
      B ← 10 (1 (2 3 4 5))
      ρB
2
      B[1]
10
      ρ⊃B[2]
2
```

In general, the following identities hold:

```
        A B   ↔   (⊂A),⊂B
      A B C   ↔   (⊂A),(⊂B),⊂C
```

These constructs of vector notation may be vector arguments of functions or operands of operators.

Example:

```
      1 (2 3 4) + 10
 11  12 13 14
```

COMPLEX ARITHMETIC

Complex arithmetic is achieved by considering all numbers to be elements of the complex number field, and defining all arithmetic on complex numbers.

A complex number constant may be represented in three ways:

1. real and imaginary parts separated by the letter *J*

2. magnitude and angle in radians separated by the letter *R*

3. magnitude and angle in degrees separated by the letter *D*

Thus, the number expressed conventionally as 3+4i may be written as 3*J*4. The number i (the square root of ¯1) may be written as either 0*J*1, 1*D*90, or 1*R*1.5707963267948965.

The *J* form of representation is the default for un-formatted output of complex numbers.

Either or both parts of a complex number constant may be specified in scaled notation. For example, 1.2*E*3*J*¯4*E*¯2 is the same as 1200*J*¯0.04, and 8*E*3*D*1*E*2 is the same as 8000*D*100.

SYSTEM FUZZ

Some primitive functions and operators (like Greater) will treat a complex number as real if the greater of the absolute values of the imaginary part and the tangent of the angle is less than approximately 1E¯13.

Some primitive functions and operators (like Index) will treat a real number R as whole (non-fractional) if the the fractional part of the number is less than approximately 1E¯13×1⌈|R.

Some primitive functions and operators (like Not) will treat a complex number as logical (either 0 or 1) if the distance between it and 0 or 1 on the complex plane is less than approximately 1E¯13.

System fuzz is not related to comparison tolerance (⎕CT), and cannot be set.

DISPLAY OF ARRAYS

Simple scalars and vectors are displayed in a single line. If an element in a simple vector is a number, then that number will be separated from adjacent elements by one blank.

Example:

```
      0 '*' 123 '⎕' '∆' 45 6 '∘'
0 * 123 ⎕∆ 45 6 ∘
```

Simple matrices are displayed in rectangular planes. All the elements in a given column of the matrix are displayed in the same format. Different columns may have different formats and different widths. If a column in a simple matrix contains a number, then that column will be separated from adjacent columns by one blank.

Example:

```
      2 5 ⍴ '*' '⎕' '∆' 123 45 '∘' '∇' 6 7 8
*⎕ ∆ 123 45
∘∇ 6   7  8
```

Simple multi-dimensional arrays are displayed in rectangular planes. Planes of a 3-dimensional array are displayed with an intervening blank line. Multi-dimensional hyperplanes of an array with more than three dimensions are each displayed with an intervening blank line. The cumulative effect is to separate the display of higher dimensions with an increasing number of blank lines.

Example:

```
      2 2 2 3 ρ 1
1 1 1
1 1 1

1 1 1
1 1 1


1 1 1
1 1 1

1 1 1
1 1 1
```

If a column in a simple multi-dimensional array contains a number, then that column will be separated from adjacent columns in all planes by one blank. All the elements in corresponding columns of the planes are displayed in the same format.

Example:

```
      2 3 4 ρ '****⎕⎕⎕⎕ΔΔΔΔooo',123,'∇∇',4 5,'??',56 7
**  *   *
⎕⎕  ⎕   ⎕
ΔΔ  Δ   Δ

oo  o 123
∇∇  4   5
?? 56   7
```

Empty arrays of rank greater than 1 may display on 0 or several lines, depending on their shape.

Leading zeros to the left of a decimal point, and trailing zeros to the right of a decimal point are suppressed in the display of numbers. A single zero before a decimal point is not considered a leading zero.

Example:

```
      00123000 00.0123000
123000 0.0123
```

The precision with which numbers are displayed is controlled by the the system variable Printing Precision (⎕*PP*). Its default value is 10 digits.

Example:

```
      2.718 3.141592653589793 0.000000000001
2.718 3.141592654 1E¯12
```

The precision with which numbers are stored internally is always the maximum that the implementation permits (at least 16 digits). All available precision will always be displayed if ⎕*PP* is set to 18.

Simple matrices and multi-dimensional arrays containing numbers that require decimal points, scaled form, or complex notation are displayed with all elements in a column in the same format. Decimal points and complex number separators align in columns. The columns are formatted independently.

Example:

```
      2 5 ρ 1 12.3 ¯10 345 6J7 0.1 0.12 1E23 1J2 345J6
1   12.3  ¯1E1  345    6J7
0.1  0.12  1E23   1J2 345J6
```

Some simple arrays containing complex numbers may be displayed in a form not suitable for input. That is, the complex number separators (*J*, *R*, or *D*) in each column will be aligned at the cost of possibly separating paired real and imaginary parts.

Example:

```
      4 4 ρ 0 1 2J3 4J5.6 7.8J9
0       1       2  J3  4J5.6
7.8J9   0       1      2J3
4  J5.6 7.8J9   0      1
2  J3   4  J5.6 7.8J9  0
```

The display notation for complex numbers (*J*, *R*, or *D*) is controlled by the the system variable Format Control (⎕*FC*). The default display of complex numbers in *J* notation is to ignore the real or imaginary part if it is less than the other by more than ⎕*PP* orders of magnitude.

Example:

```
      2J3E45 3E45J2
0J3E45 3E45
```

The display of complex numbers in *R* or *D* polar notation is to ignore the phase angle if its magnitude is less than 10*-⎕*PP*.

Example:

```
      1R1E¯9 1R1E¯11 1R¯1E¯11
1R1E¯9 1 1
```

if ⎕*PP* is 10 and ⎕*FC* is set for *R* notation formatting.

The displays of simple arrays are not indented. The displays of non-simple arrays (and non-simple items within an array) are indented one space, and they also include a trailing blank.

Example:

```
      3 2 ρ 1 2 3,(⊂4 5 6),7,⊂⊂8 9
 1      2
 3  4 5 6
 7   8 9
```

For clarity, the preceding display is repeated with each of the embedded blanks replaced by a comma:

```
,1,,,,,,2,
,3,,4,5,6,
,7,,,8,9,,
```

Character scalars or vectors in a numeric column of an array will be displayed like numeric integer scalars with the same number of digits.

Example:

```
      4 3ρ'ONE' 'TWO' 'THREE' 1111 22 3 ¯4 5 6.6 7 8.9 '?'
  ONE TWO   THREE
 1111  22       3
   ¯4   5       6.6
    7   8.9     ?
```

For clarity, the preceding display is repeated with each of the embedded blanks replaced by a comma:

```
,,ONE,TWO,,,THREE,,,
,1111,,22,,,,,,,3,,,
,,,¯4,,,5,,,,,,,6.6,
,,,,7,,,8.9,,,,,?,,,
```

Character scalars or vectors in a numeric column of an array which is displayed in scaled form will be aligned with the multiplier.

Example:

```
      4 2ρ'SOME' 'MORE' 1.2E13 3 '□□' 6.678E20 7 '?'
 SOME     MORE
  1.2E13 3.000E0
   □□    6.678E20
  7.0E0      ?
```

For clarity, the preceding display is repeated with each of the embedded blanks replaced by a comma:

```
,SOME     MORE
,,1.2E13,3.000E0,,
,,,□□,,,,6.678E20,
,,7.0E0,,,,,,?,,,,
```

Character scalars or vectors in a column which contains no numbers will be left adjusted.

Example:

```
      3 4⍴'ONE' 1111 22 3 'TWO' ¯4 5 666 'THREE' 7 8.9 '?'
 ONE   1111 22     3
 TWO     ¯4  5   666
 THREE    7  8.9   ?
```

For clarity, the preceding display is repeated with each of the embedded blanks replaced by a comma:

```
,ONE,,,1111,22,,,,,3,
,TWO,,,,,¯4,,5,,,666,
,THREE,,,,7,,8.9,,,?,
```

Other non-simple arrays are presented in a display which contains embedded blanks according to the ranks of the adjacent items. The number of embedded blanks is one less for character items than for other items.

Example:

```
      1 2 'MORE' (3 4) (2 2⍴⍳4) 5
 1 2 MORE  3 4   1 2   5
                 3 4
```

For clarity, the preceding display is repeated with each of the embedded blanks replaced by a comma:

```
,1,2,MORE,,3,4,,,1,2,,,5,
,,,,,,,,,,,,,,,,,3,4,,,,,
```

For more details about array display, refer to the discussion of the primitive monadic Format function on page 70, and the Printing Width system variable on page 219. The structure of a nested array may be studied in detail by using the *DISPLAY* workspace described in "Appendix E. Supplied Workspaces" on page 325.

EXPRESSIONS

An expression is a sequence of one or more syntactic tokens, which may be symbols or names representing arrays (constants or variables), functions, or operators. An expression usually indicates one or more operations to be performed.

If an expression produces an array, then it is called an array expression. An array expression is either:

1. a constant

2. a variable

3. a function together with its arguments (an argument is also an array expression)

If an expression produces a function, then it is called a function expression. A function expression is either:

1. a function

2. an operator together with its operands (an operand is also a function expression or an array expression)

Certain expressions produce neither arrays nor functions. An array or a function expression may be enclosed within parentheses.

Evaluation of an expression proceeds from right to left, unless modified by parentheses. If an expression results in an array value which is not assigned to a name, then that array value is displayed.

Example:

```
      A ← 8-3-1
      8-3-1
6
```

The order of execution may be modified by parentheses.

Example:

```
      (8-3)-1
4
```

Either blanks or parentheses are needed to separate names of adjacent constants, variables, defined functions, and defined operators. If *F* is a function, then:

```
      F 2   ↔   F 2
      F 2   ↔   F(2)
      F 2   ↔   (F)2
      F 2   ↔   (F)(2)
```

More than one blank or parenthesis between names is redundant, but permitted:

```
      F 2   ↔   F   2
      F 2   ↔   F (2)
      F 2   ↔   (F) 2
      F 2   ↔   (F) (2)
```

```
F 2  ←→  F   2
F 2  ←→  F((2))
F 2  ←→  ((F))2
F 2  ←→  ((F))((2))
```

Blanks or parentheses are not needed to separate names and primitive functions or operators, but they are permitted:

```
-2  ←→  - 2
-2  ←→  -(2)
-2  ←→  (-)2
-2  ←→  (-)(2)

-2  ←→  -   2
-2  ←→  -  (2)
-2  ←→  (-) 2
-2  ←→  (-) (2)

-2  ←→  -((2))
-2  ←→  ((-))2
-2  ←→  ((-))((2))

1+.×2  ←→      1 + . × 2
1+.×2  ←→    1   +   .   ×   2
1+.×2  ←→  (1)(+)(.)(×)(2)

A←1  ←→  A ← 1
A←1  ←→  A   ←   1
```

STATEMENTS

A statement is a line of characters which is intended to be understood by the APL2 system. It may be composed of

1. a label (which must be followed by a colon)

2. an expression (which may be composed of other expressions)

3. a comment (which must start with a ⍝)

Each of the three parts is optional, but if present, they must be in the order given. Everything in a statement to the right of the first comment symbol (⍝) that is not part of a character constant is a comment. Blanks adjacent to the ⍝ on either side are significant. This permits the texts of comments to be aligned in defined functions and operators. Refer to "Appendix A. Further Examples" on page 313 for an example.

Examples:

```
      2+3
5

      LABEL:2+3
5

      2+3⍝COMMENT
5

      LABEL:2+3  ⍝  COMMENT
5
```

FUNCTIONS

A function is an operation which takes either zero, one, or two arrays as explicit arguments and may explicitly produce an array as a result. It may be either:

DYADIC   defined for a left and a right argument

MONADIC   defined for a right argument, but not a left argument

NILADIC   defined for no arguments (A niladic function may not be used as the function operand of an operator.)

The number of arguments for which a function is defined is called its valence.

The name of a non-niladic function is ambi-valent. That is, it potentially represents both a monadic and a dyadic function, but both functions may or may not be defined. The function (either the monadic or the dyadic definition) intended in any usage is determined from syntactical context.

An ambi-valent function name may be used in the context of either a monadic or a dyadic function. If the context calls for a function definition which does not exist, then an error will occur. A function which has a monadic definition, but which does not have a dyadic definition, is strictly monadic. It may not be used in the context of a dyadic function.

Functions have long scope on the right. That is, a function's right argument is the result of the entire expression on its right which produces an array. A dyadic function has short scope on the left. That is, a function's left argument is the array on its left. The effect of these rules can be seen in the following example where the redundant parentheses are shown:

```
            8-3-1   ↔  8-(3-1)
      1 2-3 4-8 9   ↔  (1 2)-((3 4)-(8 9))
```

DEFINED FUNCTIONS

Functions may be defined with the system function ⎕FX, or with the system editor (see "The APL2 Default Editor" on page 291 or "The APL2 Extended Editor" on page 295). The header of such a definition must specify the name of the function, the argument(s) of the function, and the (optional) result (see "Function and Operator Definition" on page 275).

A dyadic defined function is ambi-valent. That is, its left argument is not required when the function is called in context. If such a dyadically defined function is called without a left argument, then the left argument will be undefined (will have no value) inside the function.

Example:

```
    ∇ Z←L F R
[1]   Z←⎕NC 2 1⍴'LR'
    ∇

      F 1
0 2

      1 F 1
2 2
```

⎕NC is described in "System Variables" on page 203.

If a function is defined with a right argument, but without a left argument, then the function is strictly monadic, and a dyadic call in context will cause a *VALENCE ERROR*.

Example:

```
    ∇ Z←G R
[1]   Z←2×R
    ∇

      G 1
2

      1 G 1
VALENCE ERROR
      1 G 1
      ∧ ∧
```

Niladic defined functions do not take arguments, and are not in the function domain of operators. If a niladic function produces an array result, then it more closely resembles a variable in context.

A defined function need not produce an explicit result.

OPERATORS

An operator is an operation which takes at least one function, and possibly another function or array, as operands, and produces a new function called a derived function. Operators are not ambi-valent. A particular operator is either monadic or dyadic, but not both. That is, a dyadic operator (one defined for two operands) may not be used with only one operand. The derived function produced by an operator may be ambi-valent.

The left operand of an operator must be a function. Thus, the only operand of a monadic operator must be a function. The right operand of a dyadic operator may be either a function or an array.

Operators have higher precedence than functions. Operators have long scope on the left. That is, an operator's left operand is the longest function expression on its left. The left function operand of an operator is terminated with either:

1. the end of the expression

2. the rightmost of two consecutive functions

3. a function which has an array to its left

A dyadic operator has short scope on the right. That is, an operator's right operand is the single function or array on its right. The effect of these rules can be seen in the following examples, where the redundant parentheses are shown:

```
          ρρ¨¨A   ↔   ρ(((ρ¨)¨)A)
      A+B+.×.*C+D   ↔   A+(B((+.×).*)(C+D))
    A+B+.(×.*)C+D   ↔   A+(B(+.(×.*))(C+D))
```

Parentheses may be placed around functions or derived functions.

DEFINED OPERATORS

Operators may be defined with the system function Fix (⎕*FX*), or with the system editor (see "The APL2 Default Editor" on page 291 or "The APL2 Extended Editor" on page 295). Defined operators are specified by giving the definition of the derived function. The header of such a definition must specify:

1. the name of the operator

2. the operand(s) of the operator

3. the argument(s) of the derived function

4. the explicit result (optional)

The name and operands of a defined operator are enclosed in parentheses in the header (see "Function and Operator Definition" on page 275).

Example:

```
     ∇ Z←(F REDUCE) R
[1]    Z←F/ R
     ∇

       +REDUCE 1 2 3
6
```

In this example, *REDUCE* is the name of the operator. *F* is the operand of the monadic operator *REDUCE*. *R* is the argument of the monadic derived function (*F REDUCE*). *Z* is the result of the derived function.

Example:

```
     ∇ Z←L (F DOT G) R
[1]    Z←L F.G R
     ∇

       1 2 +DOT× 3 4
11
```

In this example, *DOT* is the name of the operator. *F* and *G* are the operands of the dyadic operator *DOT*. *L* and *R* are the arguments of the dyadic derived function (*F DOT G*). *Z* is the result of the derived function.

Defined operators are not ambi-valent, but their derived functions may be. The left operand of a defined operator must be a function. Thus, the only operand of a monadic defined operator must be a function. The right operand of a dyadic defined operator may be either a function or an array.

Monadic and dyadic operators may each produce either monadic or dyadic derived functions. The argument(s) of a derived function must be arrays. The derived function produced by a defined operator need not produce an explicit result.

For more examples of defined operators, refer to the *EXAMPLES* workspace described in "Appendix E. Supplied Workspaces" on page 325.

LOCKED FUNCTIONS AND OPERATORS

A defined function or operator may be locked by opening or closing its definition with ⍫ in the APL2 Default Editor or in the APL2 Extended Editor, or with the dyadic system function ⎕*FX*. A locked function has the following execution properties:

1. It cannot be displayed or edited, and its canonical representation is a matrix with shape 0 0.
2. It cannot be suspended, just as primitive functions cannot.
3. Weak interrupts will be ignored during its execution.
4. Any non-resource error within its scope will be converted into a *DOMAIN ERROR* in the invoking expression.

SELECTIVE SPECIFICATION

Selected elements in a named array variable may be given values while leaving the shape of the array and unselected elements unchanged. This is called selective specification. It is accomplished by Bracket Indexing, and other functions which deal only with the positions of elements, or of an item in an array. Functions that are permitted in selective specification are shown in Figure 1 on page 23.

The right-most name (not in brackets) encountered in an expression on the left of an assignment symbol is treated as an array of locations within the named array. The result of the expression may be a subset or re-arrangement (or both) of the selected locations. It is these selected locations which receive new values.

```
           V[C]  ←  X

           (, V)  ←  X
           (⌽ V)  ←  X
           (⊖ V)  ←  X
           (⍉ V)  ←  X
           (⊃ V)  ←  X
        (,[A] V)  ←  X
        (⌽[A] V)  ←  X
         (C ⍴ V)  ←  X
         (C ⌽ V)  ←  X
         (C ⊖ V)  ←  X
         (C ⍉ V)  ←  X
         (C ⊃ V)  ←  X
         (C ↑ V)  ←  X
         (C ↓ V)  ←  X
         (C / V)  ←  X
         (C ⌿ V)  ←  X
         (C \ V)  ←  X
         (C ⍀ V)  ←  X
      (C ⌽[A] V)  ←  X
      (C ↑[A] V)  ←  X
      (C ↓[A] V)  ←  X
      (C /[A] V)  ←  X
      (C \[A] V)  ←  X
         (C ⌷ V)  ←  X
      (C ⌷[A] V)  ←  X
```

Notes:
- *V* is the name of an array being selectively specified.
- *X* is an array of new elements for *V*.
- *A* is a specification of axes in *V*.
- *C* is a simple integer array.

Figure 1. Selective Specification Functions

Examples:

```
      V ← 1 2 3 4 5 6

      V[2] ← 10
      V
1 10 3 4 5 6

      V[5 4] ← 20 30
      V
1 10 3 30 20 6
```

```
      M ← 3 3ρι9
      (1 1⍉M) ← 100 200 300
      M
100   2   3
  4 200   6
  7   8 300
```

Several functions may be applied in selective specification.

Example:

```
      V ← 3 3ρ1 2 3 4 5 6 7 8 9
      V
1 2 3
4 5 6
7 8 9

      (4↑,⍉V) ← 10 20 30 40
      V
10 40  3
20  5  6
30  8  9
```

Scalars being selectively assigned to a non-scalar array of locations will be replicated as necessary.

Examples:

```
      V ← '1.23.456'

      V[2 5] ← ':'
      V
1:23:456

      ((':'=V)/V) ← '/'
      V
1/23/456
```

In selective specification of elements, dimensions of length one are ignored on either side of the assignment arrow. The remaining dimensions which are not of length one must agree in rank and length, so that multiple elements may be selectively specified at a time.

Examples:

```
      V ← '1.23.456'

      V[2 5] ← 2 1ρ';'
      V
1;23;456
```

```
      V[2 1ρ2 5] ← 2ρ','
      V
1,23,456
```

In selective specification of an item, all dimensions on the right side of the assignment arrow are significant. Only one item may be selectively specified at a time. An item of an element, or an item of an item may be selectively specified. An element of an item may not be selectively specified.

Example:

```
      V ← (2 3ρ'HERYOU')(2 2ρ'HEME')
      V
 HER  HE
 YOU  ME

      (2⊃V) ← 2 1ρ'US'
      V
 HER  U
 YOU  S

      (2⊃V) ← 1 2ρ'US'
      V
 HER  US
 YOU
```

Nested indices are not permitted with Bracket Indexing in selective specification.

The result of a selective specification is the array being specified.

Example:

```
      M ← 3 4ρ1 2 3 4 5 6 7 8 9 10 11 12
      M[2;] ← M[3;] ← 0
      M
1  2 0  4
0  0 0  0
9 10 0 12
```

SHARED VARIABLES

Shared variables in APL2 permit interfacing to other systems, subsystems, devices, the APL2 environment, and auxiliary processors. Each share is bilateral (each shared variable has two owners), although multiple variables may be shared simultaneously. Variables may be explicitly shared between:

1. two active APL workspaces

2. an active APL2 workspace and an auxiliary processor

At any instant, a shared variable has only one value -- that last assigned to it by one of its owners.

A shared variable is syntactically indistinguishable from an ordinary variable. It may be both set and referenced. System variables are shared variables which are automatically shared with the APL2 system.

Several of system functions are available for manipulating shared variables and their protocols, through which a wide variety of effects can be achieved. Refer to "System Functions" on page 181.

ATTRIBUTES OF PRIMITIVE FUNCTIONS

PERVASIVENESS

Some of the APL2 primitive functions are *pervasive*, while others are not. All pervasive functions are *scalar* functions, but a scalar function is not necessarily a pervasive one.

Scalar and pervasive functions have the following properties:

1. Monadic Scalar Functions

   a. The function is applied independently to each element in its argument.

      Example:

      ```
      -1 2  ←→  ¯1 ¯2
      ```

2. Dyadic Scalar Functions

   a. A scalar argument will be extended (replicated) to conform to the shape of the other argument.

      Example:

      ```
      1 2 + 3  ←→  4 5
      ```

   b. The function is applied independently to each corresponding pair of elements in its arguments.

      Example:

      ```
      1 2 + 3 4  ←→  4 6
      ```

Pervasive functions have the properties of scalar functions at all levels of array nesting, not just at the top level. Pervasive functions have the following additional properties:

1. Monadic Pervasive Functions

   a. The function produces a result with a structure identical to that of its argument.

   b. The function is applied independently to each simple scalar in its argument.

Example:

```
        -1 (2 3)  ↔  ¯1 (¯2 ¯3)
```

c. If applied to an empty argument, the function produces its argument unchanged.

2. Dyadic Pervasive Functions

   a. The function produces a result with a structure identical to that of its arguments (after any scalar extensions).

   b. The function is applied independently to corresponding pairs of simple scalars in its arguments (after any scalar extensions).

   Example:

   ```
           1 (2 3) + 3 (4 5)  ↔  4 (6 8)
   ```

   c. If a simple scalar corresponds to a non-simple scalar in its arguments, then the function is applied between the simple scalar and the items of the non-simple scalar.

   Example:

   ```
           1 + ⊂2 3  ↔  ⊂3 4
   ```

   d. If applied between empty arguments, the function produces a composite empty structure resulting from any scalar extensions, and showing preference for the right argument if data types differ.

   Example:

   ```
           (0ρ⊂' ' (0 0)) = 0ρ⊂0 0  ↔  0ρ⊂0 (0 0)
   ```

PERVASIVE FUNCTIONS

The primitive pervasive functions are shown in Figure 2. Some symbols (~ ? ∊) denote monadic pervasive functions, but their dyadic forms are not pervasive.

Symbol	Monadic	Pg	Dyadic	Pg
+	Conjugate	36	Add	78
-	Negative	40	Subtract	93
×	Direction	36	Multiply	87
÷	Reciprocal	41	Divide	81
|	Magnitude	39	Residue	99
⌊	Floor	38	Minimum	86
⌈	Ceiling	35	Maximum	85
*	Exponential	37	Power	91
⍟	Natural Log	40	Logarithm	84
○	Pi Times	41	Circular	80
!	Factorial	37	Binomial	79
~	Not	40	{note[1]}	
?	Roll	42	{note[1]}	
∊	Type	42	{note[1]}	
∧			And	79
∨			Or	91
⍲			Nand	87
⍱			Nor	88
<			Less	83
≤			Not Greater	89
=			Equal	82
≥			Not Less	90
>			Greater	83
≠			Not Equal	88

Note:
All dyadic forms may take an axis.
[1] The dyadic form is not pervasive.

Figure 2. Primitive Pervasive Functions

PERVASIVE FUNCTION AXES

Any of the primitive dyadic pervasive functions may be applied with an axis specification as follows:

```
      Z ← L F[A] R
```

where *A* is a simple scalar or vector selection of axes, not containing repetitions, and satisfying the following expressions:

```
      (⍴,A) = (⍴⍴L)⌊⍴⍴R
      ∧/ A ∊ ⍳(⍴⍴L)⌈⍴⍴R
```

Either *L* must be a sub-array of *R*, or *R* must be a sub-array of *L*. An axis specification modifies the behavior of the pervasive function *F* at the top level of application to its arguments. At deeper levels, the behavior of the function *F* is unchanged.

Examples:

```
      10 20 +[1] 2 3⍴1 2 3 4 5 6
11 12 13
24 25 26
```

to add a vector to each column of a matrix.

```
      10 20 30 +[2] 2 3⍴1 2 3 4 5 6
11 22 33
14 25 36
```

to add a vector to each row of a matrix.

The order of multiple axes used with dyadic pervasive functions does not matter.

Examples:

```
      (2 3⍴⍳6) +[1 2] 2 3⍴10×⍳6
11 22 33
44 55 66

      (2 3⍴⍳6) +[2 1] 2 3⍴10×⍳6
11 22 33
44 55 66
```

Axis numbers *A* are origin dependent. Any empty vector is treated as an empty numeric vector, and is acceptable for an axis specification. See also "Bracket Axis Operator" on page 168.

NON-PERVASIVE FUNCTIONS

The primitive non-pervasive functions are shown in Figure 3 on page 32. Some symbols (~ ? ∊) denote dyadic non-pervasive functions, but their monadic forms are pervasive.

Non-pervasive functions have the following properties:

1. A non-pervasive function produces a result with a structure in general different from that of its arguments.

2. Arguments of a dyadic non-pervasive function are not necessarily extended.

The primitive non-pervasive functions are divided into classes:

STRUCTURAL Produces an array of data type similar to the right argument (generally dependent on its rank, shape, or nesting, but independent of the elements within it), possibly under control of a left argument.

SELECTION Produces an array of data type similar to the right argument, which is a subset, cross section, or re-organization of its elements, possibly under control of a left argument.

SELECTOR Produces a simple logical or integer array which is a map or set of indices of its right argument, possibly under control of a left argument.

MIXED Produces an array of data type similar to the right argument dependent on the elements within it, and possibly dependent on the elements within a left argument.

TRANSFORMATION Produces an array of data type independent of that of the right argument, possibly under control of a left argument.

MISCELLANEOUS May not take explicit arguments, or may not have the syntax of a function. Miscellaneous functions are not in the function domain of operators.

Some symbols (ρ ⊃ ⌷) denote non-pervasive functions which have monadic and dyadic uses in different classes.

NON-PERVASIVE FUNCTION AXES

Some non-pervasive functions may take an optional numeric axis specification in brackets. The axis numbers usually specify along which axes of one or both arguments the function is to be applied. An axis specification for a primitive non-pervasive function modifies the function's behavior in a manner dependent upon the particular function.

An axis specification may be a numeric scalar or vector, while the range of permitted values is determined by the particular function and arguments with which it is used. An axis specifi-

Class	Sym	Monadic	Pg	Dyadic	Pg
Structural	ρ	{note[1]}		Reshape	99
	,	Ravel []	49	Catenate []	95
	⌽	Reverse []	52	Rotate []	100
	⊖	Reverse []	52	Rotate []	100
	⍉	Transpose	53	Transpose	102
	⊂	Enclose []	46		
	⊃	Disclose []	43	{note[1]}	
	∪	Unite	54		
Selection	⊃	First	56	Pick	114
	↑			Drop []	105
	↓			Take []	118
	/			Replicate []	115
	⌿			Replicate []	115
	\			Expand []	107
	⍀			Expand []	107
	⌷	{note[1]}		Index []	109
	~	{note[1]}		Without	120
Selector	⍳	Interval	61	Index of	131
	⌷	Index set	60	{note[1]}	
	∩	Unique []	61		
	∊	{note[1]}		Member	131
	⍋	Grade Up	59	Grade Up	129
	⍒	Grade Down	57	Grade Down	128
	?	{note[1]}		Deal	122
	⍷			Find []	122
	⍸			Find Ind. []	125
Mixed	⊤			Encode	132
	⊥			Decode	132
	⌹	Mat. Inv.	65	Mat. Divide	133
	⍂	Eigen	64		
	⌼	Poly. Zeros	67		
Transform.	ρ	Shape	74	{note[1]}	
	≡	Depth	68	Match	137
	⍎	Execute	69		
	⍕	Format	70	Format	138
Misc.	→	Branch	148		
	[;]			Indexing	146

Notes:
[] indicates that an axis specification is optional.
[1] This function is in another class.

Figure 3. Primitive Non-Pervasive Functions

cation which is an empty vector is treated as an empty integer vector.

A non-pervasive function used without an axis specification is usually considered a shorthand notation for some default axes of the right argument (typically ⍳⍴⍴R, or ⌈/⍳⍴⍴R). Axis numbers depend on the index origin. See also "Bracket Axis Operator" on page 168.

AMBIGUOUS SYMBOLS

Some symbols (/ ⌿ \ ⍀) denote either dyadic non-pervasive functions or monadic operators, depending upon the context in which they are found. They are called ambiguous symbols. If the object to the immediate left of an ambiguous symbol is either an array or a dyadic operator, then the symbol denotes a dyadic function. In all other cases, the symbol denotes a monadic operator (that is, if the object to the immediate left of an ambiguous symbol is either a function, a monadic operator, or another ambiguous symbol, or it is neither an array nor a dyadic operator).

Pairs of brackets may denote three things:

1. Bracket Indexing, if there is an array to the immediate left of the left bracket.

2. An axis specification, if there is a primitive monadic or dyadic function or operator to the immediate left of the left bracket, and the brackets contain no delimiting semicolons.

3. The Bracket Axis operator, if there is a monadic or dyadic function to the immediate left of the left bracket, and the brackets contain one or two delimiting semicolons.

FUNCTION PRESENTATION

This manual presents the individual primitive pervasive functions first, and then the primitive non-pervasive functions in alphabetical order within class. All the monadic functions are presented before the dyadic functions. Figure 2 on page 29 and Figure 3 on page 32 show the function names and the pages where they can be found. The names of the function symbols, and the pages where descriptions of their uses begin, can be found in Figure 18 on page 286. The names of the function symbols as well as the names of the functions can also be found in the Index.

PRIMITIVE MONADIC FUNCTIONS

The primitive monadic functions take a single array argument *R* (with possibly an axis specification *A*) and produce an array result *Z*.

PRIMITIVE MONADIC PERVASIVE FUNCTIONS

The primitive monadic pervasive functions are described as they apply to a simple scalar *R* in an arbitrary array, and produce a corresponding simple scalar *Z*. The extension of the function to each simple scalar in the argument is described in "Pervasiveness" on page 27.

Example:

```
      ⌈ 10.1 (20.2 30.3)
 11  21 31
```

If a primitive monadic pervasive function is executed on an array which contains any element outside the domain of the function, then *DOMAIN ERROR* will be reported.

Ceiling: *Z ← ⌈ R*

R may be any real or complex number. If *R* is a real number, then *Z* is the smallest integer which is not less than *R* (within the comparison tolerance). Ceiling is defined in terms of the function Floor:

```
      ⌈R  ←→  -⌊-R
```

for all *R*.

Examples:

```
      ⌈ 2
2

      ⌈ 2.3
3

      ⌈ ¯2.3
¯2
```

```
      ⌈ 1J2
1J2

      ⌈ 1.2J2.5
1J3

      ⌈ 1.5J2.5
2J2

      ⌈ 1.5J2.8
2J3
```

⎕CT is an implicit argument of Ceiling.

Conjugate: Z ← + R

R may be any real or complex number. Z is the complex number whose real part is the real part of R and whose imaginary part is the negative of the imaginary part of R.

Examples:

```
      + ¯4
¯4

      + 2.3
2.3

      + 1J2
1J¯2
```

Direction: Z ← × R

R may be any real or complex number. If R is 0 then Z is 0. If R is not 0 then Z is the complex number of magnitude one with the same phase as R.

Examples:

```
      × 0
0

      × ¯4
¯1
```

```
      × 2.3
1

      × 3J4
0.6J0.8
```

Identity:

```
      × R  ←→  R÷|R
```

for all R.

Exponential: Z ← * R

R may be any real or complex number. Z is the Rth power of the base of the natural logarithms, e, where e is approximately 2.7182818284590452.

Examples:

```
      * 0
1

      * 1
2.718281828

      * 0J1
0.5403023059J0.8414709848

      * ○0J1
¯1
```

Factorial: Z ← ! R

R may be any real or complex number except for a negative integer. Z is the Gamma function of R+1. In particular, if R is a positive integer, then Z is the product of the first R positive integers.

Examples:

```
      ! 3
6

      ! 4
24

      ! 0.5
0.8862269255

      ! 1J1
0.6529654964J0.3430658398
```

Floor: Z ← ⌊ R

R may be any real or complex number. If *R* is a real'number, then *Z* is the largest integer which is not greater than *R* (within the comparison tolerance). If *R* is positive, then *Z* is the integer part of *R*.

Examples:

```
      ⌊ 2
2

      ⌊ 2.3
2

      ⌊ ¯2.3
¯3
```

If *R* is the complex number *A*+0J1×*B* (where *A* and *B* are real), then:

```
  If   1 > (A-⌊A)+B-⌊B
  then  Z  ←→   (⌊A)+0J1×⌊B

  If   1 ≤ (A-⌊A)+B-⌊B   and   (A-⌊A) ≥ B-⌊B
  then  Z  ←→   (1+⌊A)+0J1×⌊B

  If   1 ≤ (A-⌊A)+B-⌊B   and   (A-⌊A) < B-⌊B
  then  Z  ←→   (⌊A)+0J1×1+⌊B
```

This definition preserves the relation 1 > |*R*-⌊*R*, and produces a diamond, or diagonal brick pattern in the complex plane.

Examples:

```
          ⌊ 1J2
1J2

          ⌊ 1.2J2.5
1J2

          ⌊ 1.5J2.5
2J2

          ⌊ 1.5J2.8
1J3
```

⎕*CT* is an implicit argument of Floor.

Magnitude: *Z* ← | *R*

R may be any real or complex number. If *R* is the complex number *A*+0J1×*B* (where *A* and *B* are real), then

```
          Z  ↔  ((A*2)+B*2)*0.5
```

which is the non-negative magnitude of *R*.

Examples:

```
          | 0
0

          | 3
3

          | ¯3
3

          | ¯3.4
3.4

          | ¯3J4
5
```

Identity:

```
          | R  ↔  (R×+R)*0.5
```

for all *R*.

Natural Logarithm: *Z* ← ● *R*

R may be any non-zero real or complex number. *Z* is the logarithm of *R* to the base of the natural logarithms, e, where e is approximately 2.7182818284590452.

Examples:

```
      ● 1
0

      ● 2.7182818284
1

      ● ¯1
0J3.141592654

      ● 0J1
0J1.570796327
```

Negative: *Z* ← - *R*

R may be any real or complex number. Negative is defined in terms of the function Minus:

```
      -R  ↔  0 - R
```

for all *R*.

Examples:

```
      - 3
¯3

      - 3J¯4
¯3J4
```

Not: *Z* ← ~ *R*

R may be 0 or 1. If *R* is 0, then *Z* is 1. If *R* is 1, then *Z* is 0.

Examples:

```
      ~ 0
1

      ~ 1
0
```

Pi Times: *Z* ← ○ *R*

R may be any real or complex number. *Z* is pi times *R*, where pi is approximately 3.141592653589793238.

Examples:

```
      ○ 1
3.141592654

      ○ ¯1
¯3.141592654

      ○ 0J1
0J3.141592654
```

Reciprocal: *Z* ← ÷ *R*

R may be any non-zero real or complex number. Reciprocal is defined in terms of the function Divide:

```
      ÷R  ←→  1 ÷ R
```

for all valid *R*.

Examples:

```
      ÷ 1
1

      ÷ 2
0.5

      ÷ 2J1
0.4J¯0.2
```

Roll: Z ← ? R

R may be any positive integer. Z is an integer selected randomly from the integers ⍳R, with each integer in this population having equal chance of being selected.

Examples:

```
      ? 5
1

      ? 5
4
```

⎕IO and ⎕RL are implicit arguments of Roll. A side effect of Roll is to change the value of ⎕RL.

Type: Z ← ∊ R

R may be any character, or any real or complex number. If R is a character, then Z is ' ' (a scalar blank). If R is a number, then Z is (scalar) 0.

Examples:

```
      ∊ 0
0

      ∊ 3J4
0

      ' ' = ∊ '*'
1
```

PRIMITIVE MONADIC STRUCTURAL FUNCTIONS

The primitive monadic structural functions are those that deal with the rank, shape, or nesting of an array *R*, generally independently of the elements within the array, and produce an array *Z* of similar data type.

Disclose with Axis: Z ← ⊃[A] R

R must be an array such that all non-scalar items have the same rank, which must be ρ,*A*. If all items of *R* are scalar, then *A* must be empty. *A* is a simple scalar or vector of integer axes. The items of *R* are combined into a new array *Z*, with the depth reduced by one (unless *A* is empty and *R* is simple). If items of *R* have different shapes, then they will all be extended with their corresponding fill elements (at the right) so that they conform to the shape of the largest item.

The axis specification *A* refers to axes in the result *Z*, rather than to axes in the argument *R*. The result *Z* has rank (ρρ*R*)+ρ,*A*. Disclose with Axis is the left inverse of Enclose with Axis:

```
      R  ↔  ⊃[A] ⊂[A] R
```

Examples:

```
      Z ← ⊃[1] (1 2 3)(4 5 6)
      ρZ
3 2
      Z
1 4
2 3
3 6

      Z ← ⊃[2] (1 2 3)(4 5 6)
      ρZ
2 3
      Z
1 2 3
4 5 6
```

```
      Z ← ⊃[1] ' ONCE' 'MORE'
      ⍴Z
5 2
      Z
 M
OO
NR
CE
E

      Z ← ⊃[2] ' ONCE' 'MORE'
      ⍴Z
2 5
      Z
 ONCE
MORE

      ⍴ ⊃[1] 2 3⍴⊂0⍴0
0 2 3

      ⍴ ⊃[3 2] 0⍴⊂3 2⍴0
0 2 3

      R ← (4 3⍴'ME YOUHIMHER')(4 3⍴'WE US HISOUR')
      R
 ME   WE
 YOU  US
 HIM  HIS
 HER  OUR

      Z ← ⊃[2 3] R
      ⍴Z
2 4 3
      Z
ME
YOU
HIM
HER

WE
US
HIS
OUR
```

```
      Z ← ⊃[1 3] R
      ⍴Z
4 2 3
      Z
ME
WE

YOU
US

HIM
HIS

HER
OUR
```

If all items of *R* do not have the same shape (after scalar extension), then they will be padded on the end(s) with their corresponding fill elements (⊂∊⊃*ITEM*) to give them the same shape.

Examples:

```
      Z ← ⊃[2] (1 2)(3 4 5)
      ⍴Z
2 3
      Z
1 2 0
3 4 5

      R ← ⊂'MY' 'THINGS'
      R ← R,⊂ 'HER' 'BIG' 'RED' 'HAT'
      R ← R,⊂ 'SEVERAL' 'MORE'
      R
  MY THINGS   HER BIG RED HAT   SEVERAL MORE

      ⍴ R
3

      ≡ R
3

      ⍴¨ R
 2  4  2

      ⍴¨¨ R
  2  6    3  3  3  3    7  4

      Z ← ⊃[2] R
      ⍴Z
3 4
```

```
     Z
MY      THINGS
HER     BIG    RED     HAT
SEVERAL MORE

     ρ¨ Z
2  6  2  2
3  3  3  3
7  4  7  7

     Z = ' '
0 0            0 0 0 0 0 0  1 1            1 1
0 0 0          0 0 0        0 0 0          0 0 0
0 0 0 0 0 0 0  0 0 0 0      1 1 1 1 1 1 1  1 1 1 1 1 1 1

     Z ← ⊃[2]¨ R
     ρ¨Z
3
     ρ¨Z
2 6  4 3  2 7

     Z
MY      HER  SEVERAL
THINGS  BIG  MORE
        RED
        HAT

     Z = ' '
0 0 1 1 1 1  0 0 0  0 0 0 0 0 0 0
0 0 0 0 0 0  0 0 0  0 0 0 0 1 1 1
             0 0 0
             0 0 0
```

Enclose:	*Z* ← ⊂ *R*

R may be any array. *Z* is a scalar array (rank zero) whose only item is the array *R*. The depth of *Z* is one more than the depth of *R*, unless *R* is a simple scalar (a character or a number).

Identity:

```
     ,0  ←→  ρρ⊂R
```

for all *R*.

The Enclose of a simple scalar leaves the simple scalar unchanged.

Examples:

```
      ⊂ 2
2

      ⊂ 2 3
 2 3

      ⊂ 'ME'
 ME
```

Note that the last two results are non-simple arrays, and are displayed indented one space.

Enclose with Axis: Z ← ⊂[A] R

R may be any array. *A* is a simple scalar or vector of integer axes in *R*. The set of axis specified by *A* are enclosed, forming an array *Z* of rank (⍴⍴*R*)-⍴,*A*, with items of rank ⍴,*A*. The shape of *Z* is (⍴*R*)[(⍳⍴⍴*R*)~*A*], and the shape of each of the items of *Z* is (⍴*R*)[,*A*]. The depth of *Z* is one more than the depth of *R*, unless *A* is empty and *R* is simple.

Identities:

```
       ⊂R   ↔   ⊂[⍳⍴⍴R]R
      ⊂¨R   ↔   ⊂[⍳0]R
```

for all *R*.

Disclose with Axis is the left inverse of Enclose with Axis:

```
      R  ↔  ⊃[A] ⊂[A] R
```

Examples:

```
      Z ← ⊂[1] 2 3⍴1 2 3 4 5 6
      ⍴Z
3
      Z
 1 4  2 5  3 6

      Z ← ⊂[2] 2 3⍴1 2 3 4 5 6
      ⍴Z
2
      Z
 1 2 3  4 5 6
```

```
      R ← 2 3 4ρ'LESSSOMENONEMOREMANYMOST'
      R
LESS
SOME
NONE

MORE
MANY
MOST

      Z ← ⊂[1] R
      ρZ
3 4
      ρ⊃Z
2
      Z
 LM EO SR SE
 SM OA MN EY
 NN OO NS ET

      Z ← ⊂[2] R
      ρZ
2 4
      ρ⊃Z
3
      Z
 LSN EOO SMN SEE
 MMM OAO RNS EYT

      Z ← ⊂[3] R
      ρZ
2 3
      ρ⊃Z
4
      Z
 LESS SOME NONE
 MORE MANY MOST

      Z ← ⊂[1 3] R
      ρZ
3
      ρ⊃Z
2 4
      Z
 LESS  SOME  NONE
 MORE  MANY  MOST
```

```
      Z ← ⊂[2 3] R
      ρZ
2
      ρ⊃Z
3 4
      Z
 LESS  MORE
 SOME  MANY
 NONE  MOST

      Z ← ⊂[3 2] R
      ρZ
2
      ρ⊃Z
4 3
      Z
 LSN  MMM
 EOO  OAO
 SMN  RNS
 SEE  EYT
```

Ravel: Z ← , R

R may be any array. Z is a vector of length ×/ρR whose elements are the elements of R, taken in row major order.

Examples:

```
      R ← 2 3ρ1 2 3 4 5 6
      R
1 2 3
4 5 6
      , R
1 2 3 4 5 6

      R ← 2 2 2ρ1 2 3 4 5 6 7 8
      R
1 2
3 4

5 6
7 8
      , R
1 2 3 4 5 6 7 8

      , 5 1ρ'SEVEN'
SEVEN
```

```
      R ← 1 3 2⍴'YOU' 'ME' 'WE' 'THEY' 'US' 'THEM'
      R
 YOU ME
 WE  THEY
 US  THEM
      , R
 YOU ME WE THEY US THEM
```

Raveling an array does not change its depth, except for a simple scalar.

Example:

```
      R ← 2 2⍴1 (2 2) (2 2⍴3) (3 3⍴4)

      R
   1   2 2

 3 3   4 4 4
 3 3   4 4 4
       4 4 4

      , R
 1  2 2   3 3   4 4 4
          3 3   4 4 4
                4 4 4
```

Ravel with Axis: Z ← ,[A] R

R may be any array. A is a simple scalar or vector axis specification. Z is an array whose elements are the elements of R, but reshaped according to the axes A. There are two distinct cases:

1. If A is a scalar or vector of contiguous integer axes (in increasing order) of R, then Z has those axes combined, and has rank 1+(⍴⍴R)-⍴,A.

2. If A specifies a single fractional axis of R such that A>⎕IO-1 and A<⎕IO+⍴⍴R, then Z has the corresponding position in its shape filled with a new axis of length one, and Z has rank 1+⍴⍴R.

In all cases,

```
     ,R  ←→  ,,[A] R
```

If *A* is empty, then:

```
      ,[⍳0] R  ←→  ((⍴R),1)⍴R
```

Examples:

```
      Z ← ,[⍳0] 'ONE'
      Z
O
N
E
      ⍴Z
3 1

      Z ← ,[1 2] 2 3 4⍴⍳24
      Z
 1  2  3  4
 5  6  7  8
 9 10 11 12
13 14 15 16
17 18 19 20
21 22 23 24
      ⍴Z
6 4

      Z ← ,[1.5] 3 4
      Z
3
4
      ⍴Z
2 1

      Z ← ,[0.5] 'ONE'
      Z
ONE
      ⍴Z
1 3

      Z ← ,[1.5] 'ONE'
      Z
O
N
E
      ⍴Z
3 1
```

Reverse: *Z* ← ⌽ *R*

R may be any array. *Z* is an array with the same shape as *R*, and with the elements of *R* reversed along the last axis.

Examples:

```
      R ← 3 4⍴⍳12
      R
1  2  3  4
5  6  7  8
9 10 11 12

      ⌽ R
 4  3  2 1
 8  7  6 5
12 11 10 9

      R ← 2 4⍴'WE  THEY'
      R
WE
THEY

      ⌽ R
  EW
YEHT
```

The symbol ⊖ may be used instead of ⌽ to indicate the first axis of *R* rather than the last.

Examples:

```
      ⊖ 3 4⍴⍳12
9 10 11 12
5  6  7  8
1  2  3  4

      ⊖ 2 4⍴'WE  THEY'
THEY
WE
```

Reverse with Axis: *Z* ← ⌽[*A*] *R*

R may be any array. *A* is a simple scalar or one element vector which specifies an integer axis in *R*. *Z* is an array with the same shape as *R*, and with the elements of *R* reversed along the axis specified by *A*.

Examples:

```
      ⌽[2] 2 3 4⍴⍳24
 9 10 11 12
 5  6  7  8
 1  2  3  4

21 22 23 24
17 18 19 20
13 14 15 16

      ⌽[2] 2 3 5⍴'IN   OUT  UP   DOWN LEFT RIGHT'
UP
OUT
IN

RIGHT
LEFT
DOWN
```

The symbol ⊖ may be used instead of ⌽.

Transpose (Monadic):   Z ← ⍉ R

R may be any array. Z is an array of shape ⌽⍴R, similar to R, with the order of the axes of R reversed.

Identities:

```
       R   ↔   ⍉⍉R
     ⌽⍴R   ↔   ⍴⍉R
      ⍉R   ↔   (⌽⍳⍴⍴R)⍉R
```

for all R.

Examples:

```
      R ← 3 4⍴⍳12
      R
1  2  3  4
5  6  7  8
9 10 11 12

      ⍉ R
1 5  9
2 6 10
3 7 11
4 8 12
```

```
      ⍉ 2 4⍴'WE  THEY'
WT
EH
 E
 Y

      R ← 2 3 4⍴'LESSSOMENONEMOREMANYMOST'
      R
LESS
SOME
NONE

MORE
MANY
MOST

      Z ← ⍉R
      ⍴Z
4 3 2
      Z
LM
SM
NM

EO
OA
OO

SR
MN
NS

SE
EY
ET
```

Unite: Z ← ∪ R

R may be any array. Z is a simple vector whose elements are all the simple scalars in R, taken in (nested) row major order, such that:

```
      ∪ R  ←→  ∪∪¨ R
```

for all R.

Examples:

```
      Z ← ∪ (1 2) (⍳0) (2 2⍴3 4 5 4)
      Z
1 2 3 4 5 4
      ⍴Z
6

      Z ← ∪ 'ME' '' 'YOU' (2 4⍴'THEYTHEM')
      Z
MEYOUTHEYTHEM
      ⍴Z
13

      Z ← ∪ 'ME' '' 'YOU' ('WE' ('THEY' 'THEM'))
      Z
MEYOUWETHEYTHEM
      ⍴Z
15
```

If *R* is simple, then ∪*R* is ,*R*.

Example:

```
      ∪ 2 4⍴'THEYTHEM'
THEYTHEM
```

PRIMITIVE MONADIC SELECTION FUNCTIONS

The primitive monadic selection functions are those that extract a sub-array, cross-section, re-organization, or other element selection of an array *R*, and produce an array *Z* of similar data type.

First: *Z* ← ⊃ *R*

R may be any array. If *R* is not empty, then *Z* is an array whose value is the first item of *R* taken in row major order. If *R* is empty, then *Z* is the prototype of *R* (its disclosed structure).

Identity:

```
      R  ↔  ⊃⊂R
```

for all *R*.

Examples:

```
      ⊃ 1 2 3 4 5
1

      ⊃ (1 2)(3 4 5)
1 2

      ⊃ 'ME'
M

      ⊃ 'ME' 'YOU'
ME

      ⊃ 0ρ⊂1 2
0 0

      ⊃ ⊂[1] 2 0ρ0
0 0

      ⊃ 0ρ⊂ 1 (2 3)
 0  0 0
```

PRIMITIVE MONADIC SELECTOR FUNCTIONS

The primitive monadic selector functions are those that generate indices or a map *Z* of an array *R*.

Grade Down:	*Z* ← ⍒ *R*

R must be a non-scalar non-mixed simple array (of either characters or numbers, but not both). *Z* has shape 1↑⍴*R*, and is the permutation of ⍳1↑⍴*R* that puts the sub-arrays along the first axis of *R* in non-ascending order. The indices of any set of identical sub-arrays in *R* occur in *Z* in ascending order.

Examples:

```
      ⍒ 6 8 6 7 6 8 8 9
8 2 6 7 4 1 3 5

      ⍒ 4 2⍴6 8 6 7 6 8 8 9
4 1 3 2
```

If *R* is a character array, then ⍒*R* is treated like *L*⍒*R*, where *L* is a default collating sequence. The default collating sequence array is shown in Figure 4 on page 58. It is 3-dimensional, and has shape 10 2 28. The first character in each row is a blank.

The default collating sequence array sorts an alphanumeric character vector *R* such that the ascending order is:

```
      ' AA̲aBB̲bCC̲cDD̲dEE̲eFF̲fGG̲gHH̲hII̲i
         JJ̲jKK̲kLL̲lMM̲mNN̲nOO̲oPP̲pQQ̲qRR̲r
         SS̲sTT̲tUU̲uVV̲vWW̲wXX̲xYY̲yZZ̲z0123456789'
```

Example:

```
      R ← 5 4⍴'DEALLEADDEADDEEDDALE'
      R
DEAL
LEAD
DEAD
DEED
DALE

      ⍒ R
2 4 1 3 5
```

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ0
A̲B̲C̲D̲E̲F̲G̲H̲I̲J̲K̲L̲M̲N̲O̲P̲Q̲R̲S̲T̲U̲V̲W̲X̲Y̲Z̲

                          1
abcdefghijklmnopqrstuvwxyz

                          2


                          3


                          4


                          5


                          6


                          7


                          8


                          9

```

Figure 4. The Default Collating Sequence Array

The default collating sequence array also has the property that it sorts numeric integer suffixes in rows of a matrix in numeric order.

Example:

```
      R ← 8 3ρ'X1 X10X2 X21X3 X9 X11X̲3 '
      R
X1
X10
X2
X21
X3
X9
X11
X̲3
```

```
      ⍒ R
4 7 2 8 6 5 3 1

      R[⍒R;]
X21
X11
X10
X3
X9
X3
X2
X1
```

⎕IO is an implicit argument of monadic Grade Down.

Grade Up: Z ← ⍋ R

R must be a non-scalar non-mixed simple array (of either characters or numbers, but not both). Z has shape 1↑⍴R, and is the permutation of ⍳1↑⍴R that puts the sub-arrays along the first axis of R in non-descending order. The indices of any set of identical sub-arrays in R occur in Z in ascending order.

Examples:

```
      ⍋ 6 8 6 7 6 8 8 9
1 3 5 4 2 6 7 8

      ⍋ 4 2⍴6 8 6 7 6 8 8 9
2 1 3 4
```

If R is a character array, then ⍋R is treated like L⍋R, where L is a default collating sequence. The default collating sequence array is shown in Figure 4 on page 58. It is 3-dimensional, and has shape 10 2 28. The first character in each row is a blank.

Example:

```
      R ← 5 4⍴'DEALLEADDEADDEEDDALE'
      R
DEAL
LEAD
DEAD
DEED
DALE

      ⍋ R
5 3 1 4 2
```

The default collating sequence array also has the property that it sorts numeric integer suffixes in numeric order.

Example:

```
      R ← 8 3⍴'X1 X10X2 X21X3 X9 X11X̲3 '
      R
X1
X10
X2
X21
X3
X9
X11
X̲3

      ⍋ R
1 3 5 6 8 2 7 4

      R[⍋R;]
X1
X2
X3
X9
X̲3
X10
X11
X21
```

⎕IO is an implicit argument of monadic Grade Up.

Index Set: Z ← ▯ R

R must be a simple scalar or vector of non-negative integers. Z is a simple array of integers not less than ⎕IO, and has shape R,⍴R. Z consists of all combinations of the Intervals of R (⍳¨R), such that the following identities are preserved:

```
      A  ←→  (▯ ⍴A) ▯ A
```

Example:

```
      ▯ 3
1 2 3

      ▯ ,3
1
2
3
```

```
      ▯ 2 3
1 1
1 2
1 3

2 1
2 2
2 3
```

⎕*IO* is an implicit argument of Index Set.

Interval: *Z* ← ⍳ *R*

R must be a simple scalar or one element non-negative integer vector. *Z* is a simple vector of length *R*, containing *R* consecutive ascending integers starting with ⎕*IO*.

Example:

```
      ⎕IO
1
      ⍳4
1 2 3 4
      ⎕IO ← 0
      ⍳4
0 1 2 3
```

⎕*IO* is an implicit argument of Interval.

Unique: *Z* ← ∩ *R*

R may be any array. *Z* is a logical array of the same shape as *R* containing 1 where the elements in *R* first occur (in row major order). The vector of unique elements of *R*, in the order in which they occur, is (∩,*R*)/,*R*. If *R* has no repetitions in its elements, then ∧/,∩*R* is 1.

Examples:

```
      ∩ 1 0 2 2 0 3 2 3 4
1 1 1 0 0 1 0 0 1
```

```
      R ← 'ME WE THEY THEM'
      ∩ R
1 1 1 1 0 0 1 1 0 1 0 0 0 0 0
      (∩R)/R
ME WTHY

      ∩ 'M' 'E' 'ME' 'W' 'E' 'WE'
1 1 1 1 0 1

      ∩ 3 3ρ1 0 2 2 0 3 2 3 4
1 1 1
0 0 1
0 0 1
```

⎕CT is an implicit argument of Unique.

Unique with Axis: Z ← ∩[A] R

R may be any array. A is a simple scalar or vector of integer axes in R. Z is a logical array of shape (ρR)[,A], containing 1 where sub-arrays along the axes complementary to A first occur. If A is a single axis, then the unique sub-arrays of R, in the order in which they occur, is (∩[A]R)/[A]R.

Examples:

```
      R ← 4 3ρ'ME YOUME TOO'
      R
ME
YOU
ME
TOO

      Z ← ∩[1] R
      Z
1 1 0 1
      Z /[1] R
ME
YOU
TOO

      Z ← ∩[1 2] 2 4 3ρR
      Z
1 1 0 1
0 0 0 0

      Z ← ∩[2] R
      Z
1 1 1
```

```
      Z /[2] R
ME
YOU
ME
TOO

      R← 2 5ρ0 1 0 2 0 0 0 0 0 0
      R
0 1 0 2 0
0 0 0 0 0

      Z ← ∩[2] R
      Z
1 1 0 1 0

      Z /[2] R
0 1 2
0 0 0
```

□*CT* is an implicit result of Unique with Axis.

PRIMITIVE MONADIC MIXED FUNCTIONS

The primitive monadic mixed functions are those that are not pervasive, but apply to an array *R*, and produce an array result *Z* depending upon the content of *R*.

Eigen: *Z* ← ⍂ *R*

R must be a simple square matrix of real numbers. *Z* is a simple real or complex matrix of shape 1 0+ρ*R* containing the eigenvalues and the eigenvectors of *R*. If *R* has shape *N* by *N*, then *Z* has *N*+1 rows and *N* columns. The first row of *Z* contains the eigenvalues of *R*, and the remaining rows of *Z* contain the corresponding right eigenvectors of *R*. That is, each column of *Z* contains an eigenvalue, and its corresponding right eigenvector.

Example:

```
      ⍂ 2 2ρ1 0 0 2
1 2
1 0
0 1
```

The eigenvalues *X* and the right eigenvectors *V* can be obtained by:

```
      Z ← ⍂R
      X ← Z[1;]
      V ← 1 0↓Z
```

They obey the identity:

```
      X×[2]V  ←→  R+.×V
```

The eigenvalues *X* and the left eigenvectors *V* can be obtained by:

```
      Z ← ⍉⍂⍉R
      X ← Z[;1]
      V ← 0 1↓Z
```

They obey the identity:

```
      X×[1]V  ←→  V+.×R
```

The eigenvalues and eigenvectors are computed using the "Implicit QL Algorithm" if *R* is symmetric, or the "QR Algorithm" if *R* is not symmetric. The numerical accuracy of the result is dependent upon the "condition" of the matrix of eigenvectors. In particular, accuracy may be degraded if there are repeated eigenvalues.

Matrix Inverse: *Z* ← ⌹ *R*

R must be a simple array of real or complex numbers with rank not more than 2. *Z* is a simple real or complex array with the same rank as *R*, and shape ⌽⍴*R*. If *R* is a non-singular square matrix, then *Z* is the matrix inverse of *R*. If *R* is a non-singular matrix with more rows than columns, then *Z* is a pseudo inverse of *R*, in the least squares sense.

If *R* is a scalar, then *Z* is ÷*R*. If *R* is a vector or a non-square matrix, then *Z* has other interpretations explained below.

The system variable Implicit Result (⎕*IR*) is set to the algebraic rank of *R*. If this is the same as the number of columns in *R*, then *R* is non-singular.

Identity:

```
      ⌹ R   ←→   I ⌹ R
```

for all non-singular matrices *R*, where *I* is an identity matrix of shape 2⍴1↑⍴*R*:

```
      I   ←→   (⍳1↑⍴R)∘.=⍳1↑⍴R
```

Examples:

```
      ⌹ 3 3⍴1 0 0 0 2 0 2 0 4
 1    0    0
 0    0.5  0
¯0.5  0    0.25

      ⌹ 3 3⍴1 0 0 0 2 0 2 0 0J4
1      0    0
0      0.5  0
0J0.5  0    0J¯0.25
```

If *R* is a vector, then *Z* is its image obtained by inversion in the unit circle (or sphere).

Example:

```
      ⌹ 3 4
0.12 0.16
```

If *R* is a singular matrix, and the system variable Matrix Divide Tolerance (⎕*MD*) is 0, then a *DOMAIN ERROR* will occur, and the system variable Implicit Result (⎕*IR*) will contain the algebraic rank of *R*.

If *R* is singular or has more columns than rows, and the system variable Matrix Divide Tolerance (⎕*MD*) is non-zero, then ⎕*MD* is taken to be a fuzz on the algebraic rank determination of *R*. If the magnitude of ⎕*MD* is suitable, then the system variable Implicit Result (⎕*IR*) is the algebraic rank of *R*, and *Z* is a <u>pseudo</u> <u>inverse</u> obeying the following identities:

```
    R      ↔   R+.×Z+.×R
    Z      ↔   Z+.×R+.×Z
    R+.×Z  ↔    +⍉R+.×Z
    Z+.×R  ↔    +⍉Z+.×R
```

Example:

```
      R ← 3 3⍴1 0 0 1 0 0 0 0 2
      R
1 0 0
1 0 0
0 0 2
      ⌹ R
DOMAIN ERROR
      ⌹R
      ∧∧

      ⎕MD ← 1E¯13
      ⌹ R
0.5 0.5 0
0   0   0
0   0   0.5
```

⎕*IR* is an implicit result of Matrix Inverse. ⎕*MD* is an implicit argument of Matrix Inverse. ⎕*MD* is not related to ⎕*CT*.

For information about the numerical accuracy of Matrix Inverse, refer to the description of the Matrix Divide function, on page 133.

Polynomial Zeros: Z ← ⌼ R

R must be a simple non-empty vector of real or complex numbers, and not containing leading zeros. R represents a polynomial with coefficients in decreasing order of powers (constant on the right). Z is a simple vector of shape ¯1+⍴R, containing the zeros of the polynomial R.

Expressed conventionally, if $f(x) = ax^3+bx^2+cx+d$, then R is the vector (a,b,c,d). If the result Z is the vector (p,q,r), then f(x) = (x-p)(x-q)(x-r). If R is real, and the length of R is even, then Z will contain at least one real number.

Examples:

```
      ⌼ ¯2 1
0.5

      ⌼ 2 0J1
0J¯0.5

      ⌼ 1 ¯2 1
1 1

      ⌼ 1 0 1
0J1 0J¯1

      ⌼ 1 ¯6 11 ¯6
1 2 3

      ⌼ 1 ¯20 154 ¯584 1153 ¯1124 420
1 2.000000033 1.999999967 3 5 7
```

The zeros are computed using the "Jenkins and Traub Algorithms". The accuracy of the solution depends on the "condition" of the polynomial. In particular, accuracy may be degraded if there are repeated zeros. Also, numerical roundoff may cause a pair of equal real zeros to appear as a complex conjugate pair.

PRIMITIVE MONADIC TRANSFORMATION FUNCTIONS

The primitive monadic transformation functions are those that are not pervasive, but apply to an arbitrary array *R*, and produce an array result Z with data type independent of that of their argument.

Depth: *Z* ← ≡ *R*

R may be any array. *Z* is a simple non-negative integer scalar. *Z* is 0 if either:

> *R* is a (simple scalar) number
>
> *R* is a (simple scalar) character

Z is 1 if *R* is a non-scalar simple array. That is, if either:

> *R* is not empty, and every item of *R* is a scalar character or number.
>
> *R* is empty, and the prototype of *R* is a scalar character or number.

If *R* is non-simple and non-empty, then *Z* is 1+⌈/,≡¨*R*. If *R* is non-simple and empty, then *Z* is ≡⊂⊃*R*. Thus, simple arrays, always have depth of either 0 or 1.

Examples:

```
      ≡ 1
0

      ≡ 1 1
1

      ≡ ⍳0
1

      ≡ 'X'
0

      ≡ 'ME'
1

      ≡ 1 2 'X'
1
```

Non-simple arrays always have depth greater than 1.

Examples:

```
      ≡ ⊂1 2 3
2

      ≡ ⊂'ME'
2

      ≡ 0ρ⊂1 2 3
2

      ≡ 'ME' 'YOU'
2

      ≡ 0ρ⊂'ME'
2
```

An array of non-zero depth *D* contains at least one item of depth *D*-1, and may contain other items of lesser depth.

Examples:

```
      ≡ '?' 'ME' ('YOU' 'TOO')
3

      ≡¨ '?' 'ME' ('YOU' 'TOO')
0 1 2

      ≡¨¨ '?' 'ME' ('YOU' 'TOO')
 0  0 0  1 1
```

Execute: *Z* ← ⍎ *R*

R must be a simple character vector or scalar containing only valid APL2 characters, and not containing any terminal control characters (see "The APL2 Character Set" on page 285). If *R* is any empty vector, then it is treated like an empty character vector.

R is taken to represent an APL2 expression, and is executed in the context of the statement in which it is found. *Z* is the value of the APL2 expression. If the expression has no value, then ⍎*R* has no value.

Example:

```
      ⍎ '⍳4'
1 2 3 4
```

If there is an error in the APL2 expression *R*, then the error report will have an extra two lines showing the content of *R*, and where the trouble occurred in *R*.

Example:

```
      ⍎ '⍳4.5'
DOMAIN ERROR
      ⍳4.5
      ∧
      ⍎'⍳4.5'
      ∧
```

Format (Monadic): Z ← ⍕ R

R may be any array. *Z* is a simple character array which will display identically to the display produced by *R*.

If *R* is simple, then *Z* has the same rank as *R*. If *R* is a simple character array, then *Z* is *R*. If *R* is non-simple, then *Z* is either a vector or a matrix.

Example:

```
      Z ← ⍕ 2 3⍴'ME YOU'
      ⍴Z
2 3
      Z
ME
YOU
```

Simple numeric arrays are formatted by columns.

Examples:

```
      Z ← ⍕ 2 3⍴1 23 4 567 8 9
      ⍴Z
2 8
      Z
  1 23 4
567  8 9

      Z ← ⍕ 2 3⍴1 2.3 4 567 8 9
      ⍴Z
2 9
      Z
  1 2.3 4
567 8   9
```

```
      Z ← ⍕ 2 3 4⍴⍳24
      ⍴Z
2 3 11
      Z
 1  2  3  4
 5  6  7  8
 9 10 11 12

13 14 15 16
17 18 19 20
21 22 23 24
```

Formatting a non-simple array will display its rectangular nesting and hierarchy, with rows and columns formatted independently. Numeric scalar items will be aligned by decimal point (whether shown or not) in their columns. Character scalar or vector items in columns containing numeric scalars will behave like numeric integer scalars with the same number of digits. Character scalar or vector items in non-numeric columns and all other items will be left adjusted.

The format of a non-simple array has one column each of leading and trailing blanks.

Examples:

```
      Z ← ⍕ 2 3⍴'ME' 1 'YOU' 2 'THEM' 3
      ⍴Z
2 13
      Z
 ME    1 YOU
  2 THEM   3
```

For clarity, the preceding display is repeated with each of the embedded blanks replaced by a comma:

```
,ME,,,,1,YOU,
,,2,THEM,,,3,
```

Row and column spacing is determined from the context of the adjacent items. The spacing increases with the rank of the items. The number of embedded blanks is one less for character items than for other items.

Example:

```
      Z ← ⍕ 2 3⍴'ME' 1 (2 3⍴'YOUHER') 2 'THEM' 3
      ⍴Z
4 14
      Z
 ME    1  YOU
          HER

  2 THEM    3
```

For clarity, the preceding display is repeated with each of the embedded blanks replaced by a comma:

```
,ME,,,,1,,YOU,
,,,,,,,,,,HER,
,,,,,,,,,,,,,,
,,2,THEM,,,,3,
```

Examples:

```
      Z ← ⍕ 0 1 2 (3 4) (5 6 7)
      ⍴Z
19
      Z
 0 1 2  3 4  5 6 7

      Z ← ⍕ 0 1 2 (3 4) (1 3⍴5 6 7)
      ⍴Z
1 20
      Z
 0 1 2  3 4   5 6 7

      Z ← ⍕ 5 1⍴0 1 2 (3 4) (1 3⍴5 6 7)
      ⍴Z
6 7
      Z
     0
     1
     2
   3 4

 5 6 7
```

For clarity, the preceding display is repeated with each of the embedded blanks replaced by a comma:

```
,,,,,0,
,,,,,1,
,,,,,2,
,,,3,4,
,,,,,,,
,5,6,7,
```

The definition of monadic Format is applied recursively so that non-simple items within a non-simple array appear with a leading and a trailing blank.

Example:

```
      Z ← ⍕ 1 (2 3) ((4 5)(6 7)) (8 9)
      ⍴Z
25
      Z
 1  2 3   4 5  6 7   8 9
```

For clarity, the preceding display is repeated with each of the embedded blanks replaced by a comma:

```
,1,,2,3,,,4,5,,6,7,,,8,9,
```

Example:

```
      Z ← ⍕ 'A' 'BC' ('DE' 'FG') 'HI'
      ⍴Z
19
      Z
 A BC   DE FG   HI
```

For clarity, the preceding display is repeated with each of the embedded blanks replaced by a comma:

```
,A,BC,,,DE,FG,,,HI,
```

For more examples, refer to "Display of Arrays" on page 11.

The result of monadic Format is created according to the following formal rules:

```
  If R is simple, then
        ⍴⍴Z   ←→   (⍴⍴R)⌈NOTCHAR R
    if R is simple character, then
        Z   ←→   R
    if R is simple numeric, then
        ¯1↓⍴Z   ←→   ¯1↓⍴R

  If R is non-simple, then
        ⍴⍴Z   ←→   1⌈2⌊⌈/(⍴⍴R),∪⍴¨⍴¨⍕¨R
        Z has single left and right blank pad spaces
        Z has S intermediate blank spaces between
            horizontally adjacent items A and B
            where S ← ((⍴⍴A)+NOTCHAR A) ⌈ (⍴⍴B)+NOTCHAR B
        Z has L intermediate blank lines between
            vertically adjacent items C and D
            where L ← 0⌈¯1+(⍴⍴C)⌈⍴⍴D
    if 3≤⍴⍴R, then Z may contain blank lines
              for the inter-dimension spacing
```

Where (*NOTCHAR R*) returns a 1 if *R* is not a simple character array, and a 0 otherwise:

```
     ∇ Z←NOTCHAR R
[1]    Z←1
[2]    →(1<≡R)/0
[3]    Z←' '∨.≠,∊R
     ∇
```

⎕*FC* and ⎕*PP* are implicit arguments of monadic Format.

Shape: *Z* ← ρ *R*

R may be any array. *Z* is a non-negative integer vector whose elements are the dimensions of *R*. The length of *Z* is the same as the rank of *R*. In particular, if *R* is a scalar, then *Z* is an empty vector. ρρρ*R* is always 1ρ1.

Examples:

```
      ρ 4

      ρρ 4
0

      ρρ ,4
1

      ρ 4 6
2

      ρ 3J4 6
2

      ρ 4 6 8
3

      ρ 'X'

      ρρ 'X'
0

      ρ 'ME'
2

      ρ 'INFINITY'
8

      ρ 'ME' 'YOU'
2

      ρ¨ 'ME' 'YOU'
 2  3

      ρ (1 2 3)(1 2 3 4)(1 2)
3

      ρ (1 2 3)(1 2 3 4)
2
```

```
      ρ (⍳0)(⍳0)
2
      ρ¨ (⍳0)(⍳0)
 0  0
```

PRIMITIVE DYADIC FUNCTIONS

The primitive dyadic functions take a left array argument *L*, and a right array argument *R* (with possibly an axis specification *A*) and produce an array result *Z*.

PRIMITIVE DYADIC PERVASIVE FUNCTIONS

The primitive dyadic pervasive functions are described as they apply to corresponding simple scalars *L* and *R* in the left and right array arguments, and produce a corresponding simple scalar *Z* in the result array. The extension of the function to each corresponding pair of simple scalars in the arguments is described in "Pervasiveness" on page 27.

Example:

```
      10 (20 30) + 1 (2 3)
 11  22 33
```

After any scalar extensions, the left and right arguments must conform (have the same rank and identical shapes). If they don't have the same rank, then *RANK ERROR* will be reported. If they have the same rank but different shapes, then *LENGTH ERROR* will be reported. One-element vectors extend like scalars.

If a primitive dyadic pervasive function is executed on two arrays which contain any corresponding pair of elements which are outside the domain of the function, then *DOMAIN ERROR* will be reported.

The conformability requirement, as well as scalar extension, pervades to all levels if the arguments are nested.

Ten of the primitive dyadic pervasive functions are called relational functions. They are called boolean functions when they are applied to logical arguments. These ten functions, and the results of their four possible sets of logical arguments are shown in Figure 5 on page 78. When applied to boolean functions, 0 means false, and 1 means true.

Name	Pg	F	L ← 0 0 1 1
			R ← 0 1 0 1
			Result

		0	0 0 0 0
And	79	L ∧ R	0 0 0 1
Greater	83	L > R	0 0 1 0
		L	0 0 1 1
Less	83	L < R	0 1 0 0
		R	0 1 0 1
Not Equal	88	L ≠ R	0 1 1 0
Or	91	L ∨ R	0 1 1 1
Nor	88	L ⍱ R	1 0 0 0
Equal	82	L = R	1 0 0 1
	40	~ R	1 0 1 0
Not Less	90	L ≥ R	1 0 1 1
	40	~ L	1 1 0 0
Not Greater	89	L ≤ R	1 1 0 1
Nand	87	L ⍲ R	1 1 1 0
		1	1 1 1 1

Figure 5. Boolean Functions

```
Add:    Z ← L + R
```

R may be any real or complex number. *L* may be any real or complex number. *Z* is the arithmetic sum of *L* and *R*. *Z* is a real or complex number.

Examples:

```
      0 + 2
2

      1 + ¯1
0

      1 + 3.4
4.4

      0J1 + 3.4
3.4J1
```

And: *Z* ← *L* ∧ *R*

R must be logical. *L* must be logical. *Z* is logical. *Z* is the logical And of *L* and *R*.

Examples:

```
      0 ∧ 0
0
      0 ∧ 1
0
      1 ∧ 0
0
      1 ∧ 1
1
```

Binomial: *Z* ← *L* ! *R*

R may be any real or complex number except for a negative integer. *L* may be any real or complex number except for a negative integer. *Z* is a real or complex number. Binomial is defined in terms of the function Factorial:

```
      L ! R  ←→  (!R)÷(!L)×!R-L
```

for all valid *L* and *R*.

For non-negative integer arguments, *Z* is the number of distinct ways or combinations that *L* things can be chosen from *R* things.

Examples:

```
      2 ! 5
10
      2 ! 0J2
¯2J¯1
      0J1 ! 2
0.735215582J2.205646746
```

Circular Functions: Z ← L ○ R

R may be a real or complex number. *L* must be an integer such that ¯12≤*L* and *L*≤12. *Z* is a real or complex number.

L determines which of a family of circular, hyperbolic, pythagorean, or complex numeric functions to apply to *R*. They are shown in Figure 6. Some of the complex numeric functions are available as separate primitive functions, but are also provided here for completeness.

The formulas given for ¯4○*R*, ¯8○*R*, and 8○*R* hold only for complex numbers with positive real and imaginary parts (the first quadrant). The phase of the result for other arguments is adjusted for proper placement of the cuts of the complex functions.

(-*L*) ○ *R*	*L*	*L* ○ *R*
(1-*R**2)*0.5	0	(1-*R**2)*0.5
Arcsin *R*	1	Sine *R*
Arccos *R*	2	Cosine *R*
Arctan *R*	3	Tangent *R*
(¯1+*R**2)*0.5	4	(1+*R**2)*0.5
Arcsinh *R*	5	Sinh *R*
Arccosh *R*	6	Cosh *R*
Arctanh *R*	7	Tanh *R*
-(¯1-*R**2)*0.5	8	(¯1-*R**2)*0.5
R	9	Real *R*
+*R*	10	|*R*
0J1×*R*	11	Imaginary *R*
*0J1×*R*	12	Phase *R*

Note:
All angles are in radians.

Figure 6. Circular Functions

Identities:

```
¯8○R  ↔  -8○R

R  ↔   ¯10 ¯11 +.○  9 11 ∘.○ R
R  ↔    ¯9 ¯12 ×.○ 10 12 ∘.○ R
```

for all valid *R*.

Examples:

```
      0 ○ 1
0

      1 ○ 1.5708
1

     ¯1 ○ 1
1.570796327

     ¯1 ○ 2
1.570796327J1.316957897

     9 ○ 3J4
3

    10 ○ 3J4
5

    11 ○ 3J4
4

    12 ○ 3J4
0.927295218

    ¯12 ○ ○1
¯1
```

Divide: *Z* ← *L* ÷ *R*

R may be a real or complex number. *L* may be any real or complex number. *Z* is a real or complex number. If *R*≠0, then *Z* is the numeric quotient *L* divided by *R*. If *R*=0, and *L*=0, then *Z* is 1. If *R*=0, and *L*≠0, then *L*÷*R* is a *DOMAIN ERROR*.

Examples:

```
      ¯12 ÷ 4
¯3

      2 ÷ 4
0.5

      0J12 ÷ 4
0J3

      2 ÷ 0J4
0J¯0.5
```

```
      0 ÷ 0
1

      2 ÷ 0
DOMAIN ERROR
      2÷0
      ∧∧
```

Equal: Z ← L = R

R may be any character, or real or complex number. L may be any character, or real or complex number. Z is logical (either 0 or 1).

If L and R are characters, then Z is 1 if they are the same character. If L and R are both numbers, then Z is 1 if L and R are the same within a fuzz tolerance. That is, if L and R are both real, then L is considered equal to R if (|L-R) is strictly less than or equal to approximately ⎕CT×(|L)⌈|R. If L and R are complex such that L is A+0J1×B and R is C+0J1×D, then L is considered equal to R if (|A-C)+|B-D is strictly less than ⎕CT×⌈/|A,B,C,D. This relation produces a diamond neighborhood in the complex plane. The implementation of the equality determination is approximate.

Examples:

```
      ⎕CT
1E¯13

      1 = 1
1

      1 = 1.000000000000001
1

      1 = 1.00000000001
0

      1 = 2
0

      1 = 1J0.000000000000001
1

      1 = 1J0.00000000001
0
```

⎕CT is an implicit argument of Equal.

Greater: *Z ← L > R*

R may be any real number. *L* may be any real number. *Z* is logical (either 0 or 1). If *L* is greater than *R*, and *L=R* is 0, then *Z* is 1. Otherwise *Z* is 0.

Examples:

```
      ⎕CT
1E¯13

      1 > 1
0

      1.000000000000001 > 1
0

      1.00000000001 > 1
1

      2 > 1
1
```

If either argument is a complex number, then it must be within system fuzz of a real number (see "System Fuzz" on page 11).

Example:

```
      1J0.000000000000001 > 1
0

      1J0.00000000001 > 1
DOMAIN ERROR
      1J0.00000000001>1
      ^              ^
```

⎕*CT* is an implicit argument of Greater.

Less: *Z ← L < R*

R may be any real number. *L* may be any real number. *Z* is logical (either 0 or 1). If *L* is less than than *R*, and *L=R* is 0, then *Z* is 1. Otherwise *Z* is 0.

Examples:

```
      ⎕CT
1E¯13

      1 < 1
0

      1 < 1.000000000000001
0

      1 < 1.00000000001
1

      1 < 2
1
```

If either argument is a complex number, then it must be within system fuzz of a real number (see "System Fuzz" on page 11).

Example:

```
      1 < 1J0.000000000000001
0

      1 < 1J0.00000000001
DOMAIN ERROR
      1<1J0.00000000001
      ^^
```

⎕CT is an implicit argument of Less.

Logarithm: Z ← L ⍟ R

R may be any non-zero real or complex number. If R is not 1, then L may be any non-zero real or complex number not equal to 1. If R is 1, then L may also be 1. Z is a real or complex number. Z is the base L logarithm of R. Logarithm is defined in terms of the function Natural Logarithm:

```
      L ⍟ R   ↔   (⍟R)÷(⍟L)
```

for all valid L and R.

Examples:

```
      1 ⍟ 1
1

      3J4 ⍟ 1
0
```

```
      2 ⍟ 0.5
¯1

      2 ⍟ 1
0

      2 ⍟ 2
1

      2 ⍟ 8
3

      2 ⍟ ¯2
1J4.532360142

      2 ⍟ 0J2
1J2.266180071

      0J2 ⍟ 2
0.1629839861J¯0.3693510611

      0J2 ⍟ 0J2
1
```

Maximum: *Z* ← *L* ⌈ *R*

R may be any real number. *L* may be any real number. *Z* is a real number. *Z* is the larger of the numbers *L* and *R*.

Examples:

```
      1 ⌈ 2
2

      1 ⌈ ¯2
1

      ¯1 ⌈ 2
2

      ¯1 ⌈ ¯2
¯1
```

If either argument is a complex number, then it must be within system fuzz of a real number (see "System Fuzz" on page 11).

Example:

```
      1 ⌈ 0J0.000000000000001
1

      1 ⌈ 0J0.00000000001
DOMAIN ERROR
      1⌈0J0.00000000001
      ^^
```

Minimum: *Z ← L ⌊ R*

R may be any real number. *L* may be any real number. *Z* is a real number. *Z* is the smaller of the real numbers *L* and *R*.

Examples:

```
      1 ⌊ 2
1

      1 ⌊ ¯2
¯2

      ¯1 ⌊ 2
¯1

      ¯1 ⌊ ¯2
¯2
```

If either argument is a complex number, then it must be within system fuzz of a real number (see "System Fuzz" on page 11).

Example:

```
      0 ⌊ 1J0.000000000000001
0

      0 ⌊ 1J0.00000000001
DOMAIN ERROR
      0⌊1J0.00000000001
      ^^
```

Multiply: *Z* ← *L* × *R*

R may be any real or complex number. *L* may be any real or complex number. *Z* is a real or complex number. *Z* is the arithmetic product *L* times *R*.

Examples:

```
      0 × 3
0

      1 × 3
3

      2 × 3
6

      ¯2 × 3
¯6

      1J2 × 3J4
¯5J10
```

Nand: *Z* ← *L* ⍲ *R*

R must be logical. *L* must be logical. *Z* is logical (either 0 or 1). Nand is defined in terms of the function And.

```
      L ⍲ R   ←→   ~L∧R
```

for all logical *L* and *R*.

Examples:

```
      0 ⍲ 0
1

      0 ⍲ 1
1

      1 ⍲ 1
0
```

Nor: *Z* ← *L* ⍱ *R*

R must be logical. *L* must be logical. *Z* is logical (either 0 or 1). Nor is defined in terms of the function Or.

```
      L ⍱ R  ←→  ~L∨R
```

for all logical *L* and *R*.

Examples:

```
      0 ⍱ 0
1

      0 ⍱ 1
0

      1 ⍱ 1
0
```

Not Equal: *Z* ← *L* ≠ *R*

R may be any character, or real or complex number. *L* may be any character, or real or complex number. *Z* is logical (either 0 or 1). Not Equal is defined in terms of the function Equal.

```
      L ≠ R  ←→  ~L=R
```

for all *L* and *R*.

If both *L* and *R* are logical (either 0 or 1), then this is equivalent to the logical Exclusive Or function.

Examples:

```
      '2' ≠ '1'
1

      1 ≠ '1'
1
```

```
      ⎕CT
1E¯13

      1 ≠ 1
0

      1 ≠ 1.000000000000001
0

      1 ≠ 1.00000000001
1

      1 ≠ 2
1

      1 ≠ 1J0.000000000000001
0

      1 ≠ 1J0.00000000001
1
```

⎕CT is an implicit argument of Not Equal.

Not Greater: *Z* ← *L* ≤ *R*

R may be any real number. *L* may be any real number. *Z* is logical (either 0 or 1). If *L* is less than *R*, or if *L* is equal to *R* (within fuzz), then *Z* is 1. Otherwise *Z* is 0.

Examples:

```
      ⎕CT
1E¯13

      1 ≤ 1
1

      1.000000000000001 ≤ 1
1

      1.00000000001 ≤ 1
0

      2 ≤ 1
0
```

If either argument is a complex number, then it must be within system fuzz of a real number (see "System Fuzz" on page 11).

Example:

```
      1J0.000000000000001 ≤ 1
1

      1J0.00000000001 ≤ 1
DOMAIN ERROR
      1J0.00000000001≤1
      ∧              ∧
```

If both *L* and *R* are logical (either 0 or 1), then this is equivalent to the logical Material Implication.

⎕*CT* is an implicit argument of Not Greater.

Not Less: *Z* ← *L* ≥ *R*

R may be any real number. *L* may be any real number. *Z* is logical (either 0 or 1). If *L* is greater than *R*, or if *L* is equal to *R* (within fuzz), then *Z* is 1. Otherwise *Z* is 0.

Examples:

```
      ⎕CT
1E¯13

      1 ≥ 1
1

      1 ≥ 1.000000000000001
1

      1 ≥ 1.00000000001
0

      1 ≥ 2
0
```

If either argument is a complex number, then it must be within system fuzz of a real number (see "System Fuzz" on page 11).

Example:

```
      1 ≥ 1J0.000000000000001
1

      1 ≥ 1J0.00000000001
DOMAIN ERROR
      1≥1J0.00000000001
      ∧∧
```

⎕*CT* is an implicit argument of Not Less.

Or: *Z* ← *L* ∨ *R*

R must be logical. *L* must be logical. *Z* is logical. *Z* is the logical Or of *L* and *R*.

Examples:

```
      0 ∨ 0
0
      0 ∨ 1
1
      1 ∨ 0
1
      1 ∨ 1
1
```

Power: *Z* ← *L* * *R*

If *L* is not 0, then *R* may be any real or complex number. If *L* is 0, then *R* must be a non-negative real number. *L* may be any real or complex number. *Z* is a real or complex number.

If *R* is 0, then *Z* is 1. If *R* is 1, then *Z* is *L*. If *R* is a non-negative integer, then *Z* is ×/*R*ρ*L*. In all other cases, the following generalization is preserved:

```
      L*A+B  ←→  (L*A)×L*B
```

The *N*th root of a number *L* is *L**÷*N*. In particular, the square root of a number *L* is *L**0.5. In cases where there are multiple roots, the result is the one with the least non-negative angle in the complex plane.

Examples:

```
      0 * 0
1
      2 * 0
1
```

```
      2 * 1
2

      2 * 3
8

      2 * ¯3
0.125

      ¯2 * 3
¯8

      16 * 0.5
4

      16 * 0.25
2

      ¯16 * 0.25
1.414213562J1.414213562

      2 * 0J3
¯0.486994418J0.8734050818

      0J2 * 3
0J¯8

      0J2 * 0J1
0.1599090569J0.1328269994
```

Residue: *Z* ← *L* | *R*

R may be any real or complex number. *L* may be any real or complex number. *Z* is a real or complex number.

If *L*=0, then *Z* is *R*. If *L*≠0, then *Z* is *R*-*L*×⌊*R*÷*L*. For real numbers *L* and *R*, *Z* is the remainder on dividing *L* into *R*. In particular, if *L* is positive real, then 0≤*Z* and *Z*<*L*. If *L* is negative real, then *L*<*Z* and *Z*≤0.

Examples:

```
      0 | 17
17

      1 | 17
0

      10 | 17
7

      ¯10 | 17
¯3

      10 | 8
8
      10 | 9
9
      10 | 10
0
      10 | 11
1
      10 | 12
2

      1 | 3.14159
0.14159

      0J10 | 17
¯3

      7J10 | 10J7
¯7J4

      4J6 | 7J10
3J4
```

⎕*CT* is an implicit argument of Residue.

Subtract: *Z* ← *L* - *R*

R may be any real or complex number. *L* may be any real or complex number. *Z* is a real or complex number. *Z* is the arithmetic difference *L* minus *R*.

Examples:

```
      5 - 3
2

      3 - 5
¯2

      3 - ¯5
8

      3J4 - 5
¯2J4
```

PRIMITIVE DYADIC STRUCTURAL FUNCTIONS

The primitive dyadic structural functions are those that deal with the rank or shape of an array *R*, generally independently of the items within the array, but dependent upon an array *L*, and produce an array *Z* of data type similar to *R*.

Catenate: *Z* ← *L* , *R*

R may be an array. *L* may be an array. The array *L* and the array *R* are joined along their last axis to form a (generally larger) array *Z*.

There are three cases of conformability:

> Arrays *L* and *R* may have the same shape (except possibly along the last axis), and then the last axis of *Z* has a length equal to the sum of the lengths of the last axes of *L* and *R*.
>
> One of the arguments may have a rank less than the other by one, in which case its shape is augmented to include a unit last axis, and then it must meet the requirements of the first case.
>
> Scalars will always be reshaped as needed to conform before application of the function.

With one exception, the rank of *Z* is equal to the larger of the ranks of *L* and *R*, and not more than one greater than the smaller. The exception is the case where both *L* and *R* are scalars, and then *Z* is a two element vector.

Examples:

```
      1 , 2
1 2

      1 , 2 3
1 2 3

      'RUN' , 'NY'
RUNNY
```

```
      10 , 2 3ρ1 2 3 4 5 6
10 1 2 3
10 4 5 6

      10 11 , 2 3ρ1 2 3 4 5 6
10 1 2 3
11 4 5 6

      (2 3ρ'ME YOU') , '?'
ME ?
YOU?
```

If one of *L* and *R* is empty, then the data type of *Z* will be the same as the other. If both *L* and *R* are empty, then the data type of *Z* will be the same as the data type of *R*.

Catenate with Axis: *Z* ← *L* ,[*A*] *R*

R may be an array. *L* may be an array. *A* is a simple scalar or one element vector axis specification, which may be fractional.

If *A* is an integer in the range ι(ρρL)⌈ρρR, then Catenate with Axis is like the function Catenate, except that an axis other than the last may be specified. In such a case, the rank of *Z* is equal to the larger of the ranks of *L* and *R*. If *A* is fractional, then the rank of *Z* is equal to 1 plus the larger of the ranks of *L* and *R*.

Examples:

```
      10 ,[1] 2 3ρ1 2 3 4 5 6
10 10 10
 1  2  3
 4  5  6

      10 11 12 ,[1] 2 3ρ1 2 3 4 5 6
10 11 12
 1  2  3
 4  5  6
```

```
      '*',[1] 3 2 4ρ'ME  YOU WE  THEYUS  THEM'
****
****

ME
YOU

WE
THEY

US
THEM
      '*',[2] 3 2 4ρ'ME  YOU WE  THEYUS  THEM'
****
ME
YOU

****
WE
THEY

****
US
THEM
      '*',[3] 3 2 4ρ'ME  YOU WE  THEYUS  THEM'
*ME
*YOU

*WE
*THEY

*US
*THEM
```

If A is a single fractional axis such that A>⎕IO-1 and A<⎕IO+(ρρL)⌈ρρR, then the function may be called Laminate, rather than Catenate with Axis. This specifies the formation of a new axis between two existing ones, before the first, or after the last. If A is fractional, then L and R must have the same shape, or one of them must be a scalar (which will be replicated as necessary).

Examples:

```
      10 20 ,[0.5] 1 2
10 20
 1  2

      10 20 ,[1.5] 1 2
10 1
20 2
```

```
      10 ,[0.5] 2 3⍴1 2 3 4 5 6
10 10 10
10 10 10

 1  2  3
 4  5  6

      10 ,[1.5] 2 3⍴1 2 3 4 5 6
10 10 10
 1  2  3

10 10 10
 4  5  6

      10 ,[2.5] 2 3⍴1 2 3 4 5 6
10 1
10 2
10 3

10 4
10 5
10 6

      '*',[0.5] 3 4⍴'YOU WE  THEY'
****
****
****

YOU
WE
THEY

      '*',[1.5] 3 4⍴'YOU WE  THEY'
****
YOU

****
WE

****
THEY

      '*',[2.5] 2 3⍴'YOUWE '
*Y
*O
*U

*W
*E
*
```

Reshape: *Z* ← *L* ρ *R*

R may be an array. *L* may be a simple scalar or vector of non-negative integers. *Z* is an array of shape *L* whose elements are taken sequentially from *R* in row major order, and repeated cyclically if required. If *R* is empty, then *L* must contain at least one zero. If *L* contains at least one zero, then *Z* is empty. If *L* is any empty vector, then it is treated as an empty numeric vector.

Identity:

```
      ,L   ↔   ρLρR
```

for all valid *L* and *R*.

Examples:

```
      2 3 4 ρ ⍳24
 1  2  3  4
 5  6  7  8
 9 10 11 12

13 14 15 16
17 18 19 20
21 22 23 24
      3 2 4 ρ 'ME  YOU WE  THEYUS  THEM'
ME
YOU

WE
THEY

US
THEM
      3 2 4 ρ 'MEYOUWETHEYUSTHEM'
MEYO
UWET

HEYU
STHE

MMEY
OUWE
        2 4 ρ 'MEYOUWETHEYUSTHEM'
MEYO
UWET
```

```
      2 4 ⍴ 3 1 2 5⍴'MEYOUWETHEYUSTHEM'
MEYO
UWET
```

Rotate: Z ← L ⌽ R

R may be any array. L may be a simple array of integers. Z is an array with the same shape as R.

If L is a non-negative scalar, then L elements are removed from the beginning of each vector along the last axis of R, and appended to the same vector.

If L is a negative scalar, then |L elements are removed from the end of each vector along the last axis of R, and prefixed to the same vector.

If L is a 1-element vector, then it is treated like a scalar. If L is not a scalar (or a 1-element vector), then ⍴L must be ¯1↓⍴R, and the vectors of R are treated independently according to the corresponding elements of L.

Examples:

```
      2 ⌽ 3
3

      2 ⌽ 1 2 3 4 5
3 4 5 1 2

      20 ⌽ 1 2 3 4 5
1 2 3 4 5

      1 ⌽ 3 4⍴⍳12
 2  3  4 1
 6  7  8 5
10 11 12 9

      0 1 2 ⌽ 3 4⍴⍳12
 1  2 3  4
 6  7 8  5
11 12 9 10
```

```
      (3 2⍴2 3 2 4 2 0) ⌽ 3 2 4⍴'ME  YOU WE  THEYUS  THEM'
  ME
 YOU

  WE
THEY

  US
THEM
      (-3 2⍴2 1 2 0 2 0) ⌽ 3 2 4⍴'ME  YOU WE  THEYUS  THEM'
  ME
 YOU

  WE
THEY

  US
THEM
```

The symbol ⊖ may be used instead of ⌽ to indicate the first axis of *R* rather than the last.

Example:

```
      1 ⊖ 3 4⍴'THISMANYMORE'
MANY
MORE
THIS
```

Rotate with Axis: *Z* ← *L* ⌽[*A*] *R*

R may be any non-scalar array. *L* may be a simple array of integers. *A* is a simple scalar or one element vector integer axis in *R*. *Z* is an array with the same shape as *R*, and with the elements of *R* rotated along the axis specified by *A*.

If *L* is a 1-element vector, then it is treated like a scalar. If *L* is not a scalar (or a 1-element vector), then ⍴*L* must be (⍴*R*)[(⍳⍴⍴*R*)~*A*], and the vectors of *R* are treated independently according to the corresponding elements of *L*.

Rotate with Axis is like the function Rotate, except that an axis other than the last or first may be specified.

Example:

```
      0 1 2 ⌽[2] 3 4⍴⍳12
 1  2 3  4
 6  7 8  5
11 12 9 10

      1 ⌽[2] 3 2 4⍴'ME  YOU WE  THEYUS  THEM'
YOU
ME

THEY
WE

THEM
US
```

The symbol ⊖ may be used instead of ⌽.

Transpose (Dyadic):   *Z* ← *L* ⍉ *R*

R may be any array. *L* may be a simple scalar or vector of integer axes in *R*. The number of elements in *L* must be equal to the rank of *R*. If *L* selects all axes of *R*, then *Z* is an array of shape (⍴*R*)[⍋*L*], similar to *R* with the order of the axes of *R* permuted. If there are repetitions in *L*, then *Z* is an array of rank +/∩*L*.

Identities:

```
      ⍴R   ↔   (⍴L⍉R)[,L]
      ⍉R   ↔   (⌽⍳⍴⍴R)⍉R
```

for all valid *L* and *R* where ∧/(⍳⍴⍴*R*)∊*L*.

Examples:

```
      (0⍴0) ⍉ 1
1

      2 1 ⍉ 3 4⍴⍳12
1 5  9
2 6 10
3 7 11
4 8 12
```

```
      R ← 2 3 4ρ'LESSSOMENONEMOREMANYMOST'
      R
LESS
SOME
NONE

MORE
MANY
MOST

      Z ← 1 3 2 ⍉ R
      ρZ
2 4 3
      Z
LSN
EOO
SMN
SEE

MMM
OAO
RNS
EYT

      Z ← 2 1 3 ⍉ R
      ρZ
3 2 4
      Z
LESS
MORE

SOME
MANY

NONE
MOST

      Z ← 3 1 2 ⍉ R
      ρZ
3 4 2
      Z
LM
EO
SR
SE

SM
OA
MN
EY

NM
OO
NS
ET
```

If *L* does not select all axes of *R*, then there must be repetitions in *L*. In such a case, two or more axes of *R* map into a single axis of *Z*, which is then a diagonal cross section of *R* with less rank than *R*.

Examples:

```
      1 1 ⍉ 3 3⍴⍳9
1 5 9

      1 1 2 ⍉ 3 2 4⍴'ME  YOU WE  THEYUS  THEM'
ME
THEY
```

It must always be true that ∧/(⍳⌈/0,*L*)∊*L*. ⎕*IO* is an implicit argument of dyadic Transpose.

PRIMITIVE DYADIC SELECTION FUNCTIONS

The primitive dyadic selection functions are those that extract a sub-array, cross-section, re-organization, or other selection of elements from an array *R*, and produce an array *Z* of similar data type, dependent upon an array *L*. In the cases where *L* is expected to be an array of integers, any empty array *L* is treated as an empty integer array.

Drop: *Z* ← *L* ↓ *R*

R may be any array. *L* may be a simple scalar or vector of integers. If *L* is a scalar, then it will be treated as a one element vector. If *R* is a scalar, then it will be treated as a one element array with shape (ρ*L*)ρ1. After any scalar extensions, ρ,*L* must be equal to ρρ*R*.

Z is an array with the same rank as *R* (after any scalar extension), but with its shape reduced by *L*. If *L*[*I*] (an element of *L*) is positive, then *L*[*I*] sub-arrays are removed from the beginning of the *I*th axis of *R*. If *L*[*I*] (an element of *L*) is negative, then |*L*[*I*] sub-arrays are removed from the end of the *I*th axis of *R*.

Examples:

```
      2 ↓ 1 2 3 4 5
3 4 5

      ¯2 ↓ 1 2 3 4 5
1 2 3

      1 ¯1 ↓ 4 4ρι16
 5  6  7
 9 10 11
13 14 15

      1 0 1 ↓ 3 2 4ρ'ME  YOU WE  THEYUS  THEM'
E
HEY

S
HEM
```

Drop with Axis: *Z* ← *L* ↓[*A*] *R*

R may be any non-scalar array. *L* may be a simple scalar or vector of integers. *A* may be a simple scalar or vector integer selection of axes in *R*. *A* may not contain repetitions. *Z* is an array with the same rank as *R*, but with (possibly truncated) shape (⍴*R*)[,*A*]-|*L* along axes *A*. The shape along axes not selected by *A* remains unchanged.

Conformability requires that (⍴,*A*)≤⍴⍴*R*, and that ⍴,*L* is ⍴,*A*. If *L* is a scalar, then it will be treated as a 1-element vector.

Drop with Axis is like the function Drop, except that only the selected axes are affected.

Identity:

```
      L ↓ R  ←→  L ↓[⍳⍴⍴R] R
```

Examples:

```
      (⍳0) ↓[⍳0] 1 2 3
1 2 3

      1 ↓[1] 3 4⍴⍳12
5  6  7  8
9 10 11 12

      1 ↓[2] 3 4⍴⍳12
 2  3  4
 6  7  8
10 11 12

      ¯1 ↓[1] 3 2 4⍴'ME  YOU WE  THEYUS  THEM'
ME
YOU

WE
THEY

      ¯2 ↓[3] 3 2 4⍴'ME  YOU WE  THEYUS  THEM'
ME
YO

WE
TH

US
TH
```

More than one axis may be specified. If so, then ⍴,*A* must be ⍴,*L*.

Example:

```
      1 ¯1 ↓[1 3] 3 2 4⍴'ME  YOU WE  THEYUS  THEM'
WE
THE

US
THE
```

Multiple axes specified by *A* need not be in increasing order.

Example:

```
      ¯1 1 ↓[3 1] 3 2 4⍴'ME  YOU WE  THEYUS  THEM'
WE
THE

US
THE
```

Expand: Z ← L \ R

R may be any array. *L* may be a simple logical scalar or vector. If *R* is a scalar, then it will be treated as a one element vector. If ¯1↑⍴*R* is 1, then *R* will be replicated (along the last axis) +/*L* times before application of the function. If ¯1↑⍴*R* is not 1, then it must be equal to +/*L*.

Z is an array with the same rank as *R* (but not scalar), but with the last axis expanded according to the format indicated by *L*, so that ¯1↑⍴*Z* is ⍴,*L*. Positions in *Z* where there are ones in *L* are filled with sub-arrays of *R*. Positions in *Z* where there are zeros in *L* are filled with the fill element (⊂∊⊃*R*). ¯1↓⍴*Z* is ¯1↓⍴*R*.

Identities:

```
      L \ R  ↔  ¯1 1[1+L] / R
      L \ R  ↔   (¯1+2×L) / R
```

for logical *L*.

Examples:

```
      Z ← 0 \ 0⍴0
      ⍴Z
1
      Z
0
```

```
      Z ← 1 0 0 1 1 0 \ 2
      ρZ
6
      Z
2 0 0 2 2 0

      Z ← 1 0 0 1 1 0 \ 2 4 8
      ρZ
6
      Z
2 0 0 4 8 0

      Z ← 1 0 0 1 1 0 \ (1 2)(3 4 5)(6 7 8 9)
      ρZ
6
      Z
 1 2  0 0  0 0  3 4 5  6 7 8 9  0 0
      ρ¨Z
 2  2  2  3  4  2

      Z ← 1 0 1 0 0 1 1 \ 2 4ρ'THUSTHIS'
      ρZ
2 7
      Z
T H  US
T H  IS
```

The symbol ⍀ may be used instead of \ to indicate the first axis of *R* rather than the last.

Example:

```
      Z ← 1 0 1 ⍀ 2 3ρ'HIMHER'
      Z
HIM

HER
      ρZ
3 3
```

Expand with Axis: *Z* ← *L* \[*A*] *R*

R may be any non-scalar array. *L* may be a simple logical scalar or vector. *A* may be a simple scalar or one element vector integer axis in *R*. *Z* is an array with the same rank as *R*, but with axis *A* expanded according to the format indicated by *L*, so that (ρ*Z*)[,*A*] is ρ,*L*.

Expand with Axis is like the function Expand, except that an existing axis in *R* other than the default may be specified. If

(⍴R)[A] is 1, then R will be replicated (along axis A) +/L times before application of the function. If (⍴R)[A] is not 1, then it must be equal to +/L.

Example:

```
      1 0 1 \[1] 1 3⍴'YOU'
YOU

YOU
      Z ← 1 0 1 \[1] 2 3⍴'HIMHER'
      Z
HIM

HER
      ⍴Z
3 3
      1 0 1 1 1 \[3] 3 2 4⍴'ME  YOU WE  THEYUS  THEM'
M E
Y OU

W E
T HEY

U S
T HEM
```

The symbol ⍀ may be used instead of \.

Index: Z ← L ⌷ R

R may be any array. L must be a simple array of integers not less than ⎕IO, and not greater than ⎕IO+S-1, where S is the length of the axis being indexed. If L is a scalar then it is treated like a one element vector. The columns of L correspond to the axes of R.

Conformability requires that:

```
      ¯1↑1,⍴L  ←→  ⍴⍴R
```

Z is an array of shape ¯1↓⍴L, containing elements of R as specified by indices L. Z has the same depth as R, unless R is a simple scalar, 0=¯1↑⍴R, and 2≤⍴⍴R. If L is a scalar or vector, then Z is a scalar. Each element in L specifies which element along the corresponding axis of R is to be selected for Z, similar to bracket indexing:

```
      (I,J,K) ⌷ R   ↔   R[I;J;K]
```

for simple scalars *I*, *J*, and *K*.

If *L* has more than one axis, then the rows of *L* index *R* independently, and the function is called Scatter Index.

Identity:

```
      L ⌷ R   ↔   L ⌷[;¯1↑⍳⍴⍴L;] R
```

for non-scalar *L*.

Examples:

```
      R ← 10 20 30 40
      Z ← 2 ⌷ R
      ⍴⍴Z
0
      Z
20
      Z   ↔   R[2]

      R ← 3 5⍴'VENUSEARTHPLUTO'
      R
VENUS
EARTH
PLUTO
      Z ← 1 3 ⌷ R
      ⍴⍴Z
0
      Z
N
      Z   ↔   R[1;3]
```

If *L* has rank greater than one, then items in its rows index *R* independently, and *Z* has shape ¯1↓⍴*L*.

Examples:

```
      R ← 10 20 30 40
      Z ← (3 1⍴1 2 4) ⌷ R
      ⍴Z
3
      Z
10 20 40
      Z   ↔   R[1],R[2],R[4]

      R ← 3 3⍴'HIMHERYOU'
      R
HIM
HER
YOU
```

```
      Z ← (2 2⍴1 3 2 2) ⌷ R
      ⍴Z
2
      Z
ME
      Z  ←→  R[1;3],R[2;2]

      R ← 10 20 30 40
      Z ← (2 3 1⍴1 3 2 2 2 4) ⌷ R
      ⍴Z
2 3
      Z
10 30 20
20 20 40
      Z  ←→  2 3⍴R[1],R[3],R[2],R[2],R[2],R[4]
```

⎕*IO* is an implicit argument of Index.

Index with Axis: *Z* ← *L* ⌷[*A*] *R*

R may be any array. *L* must be a simple array of integers not less than ⎕*IO*, and not greater than ⎕*IO*+*S*-1, where *S* is the length of the axis being indexed. *A* may be a simple scalar or vector integer selection of axes in *R*. *A* may not contain repetitions. *Z* is an array of shape (¯1↓⍴*L*),(~(⍳⍴⍴*R*)∊*A*)/⍴*R*, containing elements of *R* as specified by indices *L* along axes *A*. *Z* has the same depth as *R*, unless *R* is a simple scalar, and 0=⍴,*A*. Conformability requires that ¯1↑1,⍴*L* is ⍴,*A*, and that (⍴,*A*)≤⍴⍴*R*.

If the rank of *L* is less than 2, then Index with Axis is similar to Bracket Indexing with elided axes:

```
I ⌷[1] R  ←→  R[I;;]
J ⌷[2] R  ←→  R[;J;]
K ⌷[3] R  ←→  R[;;K]

(I,J) ⌷[1 2] R  ←→  R[I;J;]
(I,K) ⌷[1 3] R  ←→  R[I;;K]
(J,K) ⌷[2 3] R  ←→  R[;J;K]
```

for simple scalars *I*, *J*, and *K*.

Index with Axis is like the function Index, except that *L* supplies indexes only for the axis selected by *A*. Along the unselected axes, all indices are used, and these axes are placed last in the result.

If *L* has more than one axis, then the function is called Scatter Index with Axis.

Examples:

```
      R ← 3 3⍴'HIMHERYOU'
      R
HIM
HER
YOU

      Z ← 2 ⌷[1] R
      ⍴Z
3
      Z
HER
      Z  ←→  R[2;]

      Z ← 2 ⌷[2] R
      ⍴Z
3
      Z
IEO
      Z  ←→  R[;2]
```

If *L* has rank greater than one, then elements in its rows index *R* independently, and scatter indexing is performed along the axis selection. The unselected axes are placed last in the result.

Examples:

```
      R ← 3 3⍴'HIMHERYOU'
      R
HIM
HER
YOU

      Z ← (2 1⍴1 3) ⌷[1] R
      ⍴Z
2 3
      Z
HIM
YOU
      Z  ←→  2 3⍴R[1;],R[3;]

      Z ← (2 1⍴1 3) ⌷[2] R
      ⍴Z
2 3
      Z
HHY
MRU
      Z  ←→  2 3⍴R[;1],R[;3]
```

More than one axis may be selected.

Example:

```
      R ← 3 2 4⍴'ME  YOU WE  THEYUS  THEM'
      R
ME
YOU

WE
THEY

US
THEM
      Z ← 1 2 ⌷[1 2] R
      ⍴Z
4
      Z
YOU
      Z  ↔  R[1;2;]

      Z ← 1 2 ⌷[1 3] R
      ⍴Z
2
      Z
EO
      Z  ↔  R[1;;2]

      Z ← 1 2 ⌷[2 3] R
      ⍴Z
3
      Z
EES
      Z  ↔  R[;1;2]

      Z ← (2 2⍴1 2 3 2) ⌷[1 2] R
      ⍴Z
2 4
      Z
YOU
THEM
      Z  ↔  2 4⍴R[1;2;],R[3;2;]
```

The axes specified by *A* correspond to columns of *L*, but they need not be in ascending order.

Example:

```
      Z ← (2 2⍴2 1 2 3) ⌷[2 1] R
      ⍴Z
2 4
      Z
YOU
THEM
```

⎕*IO* is an implicit argument of Index with Axis.

Pick: Z ← L ⊃ R

R may be any array. L may be a (possibly nested) scalar or vector of integers not less than ⎕IO, and not greater than ⎕IO+S-1, where S is the length of the axis being indexed. The depth of L may be no greater than 2 (each item of L must be simple). If L has length more than 1, then R must be nested. Z is an item of R as specified by the path indices L.

If L is an empty vector, then Z is R.

Example:

```
      R ← 'ME' 'YOU' 'ALSO'
      R
 ME YOU ALSO
      (⍳0) ⊃ R
 ME YOU ALSO
```

If L is a scalar or a one element vector, then Pick is equivalent to Index followed by First, and Z is ⊃L⌷R.

Example:

```
      Z ← 2 ⊃ 'ME' 'YOU' 'ALSO'
      Z
YOU
      ⍴Z
3
```

If R is simple, then the ⊃ has no effect.

Example:

```
      2 ⊃ 1 2 3
2
```

If L has more than one item, then the function is applied recursively:

```
      L ⊃ R  ↔  (1↓L) ⊃ ⊃(⊃L) ⌷ R
      L ⊃ R  ↔  (1↓L) ⊃ (1↑L) ⊃ R
```

If L is a two element vector, then L⊃R selects an item of an item of R.

Examples:

```
      R← 'ME' 'YOU' 'ALSO'
      Z ← 2 3 ⊃ R
      Z
U
```

```
      R ← 'ME' 'YOU' 'ALSO'
      Z ← 3 1 ⊃ R
      Z
A

      R ← 2 2ρ'ME' 'AND' (2 4ρ'YOU THEM') 'ALSO'
      Z ← (2 1)(1 1) ⊃ R
      Z
Y
```

If *L* is a three element vector, then *L*⊃*R* selects an item of an item of an item of *R*.

Examples:

```
      Z ← 2 3 1 ⊃ 'ME' 'YOU' 'ALSO'
INDEX ERROR
      Z ←2 3 1⊃'ME' 'YOU' 'ALSO'
        ^    ^

      Z ← 2 3 1 ⊃ 'ME' ('YOU' 'AND' 'THEM') 'ALSO'
      Z
T
```

⎕*IO* is an implicit argument of Pick.

Replicate: Z ← L / R

R may be any array. *L* must be a simple scalar or vector of integers. *Z* is an array with the same rank as *R*, but with each sub-array along the last axis replicated according to the format indicated by *L*. ¯1↓ρ*Z* is ¯1↓ρ*R*.

If *L* is a scalar or one element vector, then it will be extended to ¯1↑1,ρ*R* elements before application of the function. If *R* is a scalar, then it will be treated as a one element vector. If ¯1↑ρ*R* is 1, then *R* will be replicated (along the last axis) +/*L*≥0 times before application of the function. If ¯1↑ρ*R* is not 1, then it must be equal to +/*L*≥0.

Non-negative elements of *L* correspond to sub-arrays of *R* along its last axis. If *L*[*I*] (an element of *L*) is non-negative, then the corresponding sub-array of *R* will be replicated *L*[*I*] times. If *L*[*I*] (an element of *L*) is negative, then *Z* is filled with |*L*[*I*] fill elements (⊂∊⊃*R*). If *L* is not extended, then ¯1↑ρ*Z* is +/|*L*.

Examples:

```
      ¯3 / 0⍴0
0 0 0

      2 / 1 2 3
1 1 2 2 3 3

      2 3 / 4
4 4 4 4 4

      1 0 2 3 / 1 2 3 4
1 3 3 4 4 4

      1 1 2 0 0 0 1 / 'MERCURY'
MERRY

      1 0 2 ¯1 3 ¯2 / 1 2 3 4
1 3 3 0 4 4 4 0 0

      0 4 0 1 / 3 4⍴⍳12
 2  2  2  2  4
 6  6  6  6  8
10 10 10 10 12
```

The symbol ⌿ may be used instead of / to indicate the first axis of R rather than the last.

Example:

```
      0 2 1 ⌿ 3 4⍴'THISMANYMORE'
MANY
MANY
MORE
```

If L is entirely logical (containing only 0, 1, or both), then the function L/ is called Compress instead of Replicate.

Replicate with Axis: Z ← L /[A] R

R may be any non-scalar array. L may be a simple scalar or vector of integers. A may be a simple scalar or one element vector integer axis in R. Z is an array with the same rank as R, but with axis A replicated according to the format indicated by L.

Replicate with Axis is like the function Replicate, except that an existing axis in R other than the default may be specified.

If L is a scalar or one element vector, then it will be extended to (⍴R)[A] elements before application of the function. If L is not extended, then (⍴Z)[A] is +/|L.

If (ρR)[A] is 1, then R will be replicated (along axis A) +/L≥0 times before application of the function. If (ρR)[A] is not 1, then it must be equal to +/L≥0.

Non-negative elements of L correspond to sub-arrays of R along axis A. If L[I] (an element of L) is non-negative, then the corresponding sub-array of R will be replicated L[I] times. If L[I] (an element of L) is negative, then Z is filled with |L[I] fill elements (⊂∊⊃R). If L is not extended, then (ρZ)[A] is +/|L.

Examples:

```
      2 3 /[1] 1 3ρ'YOU'
YOU
YOU
YOU
YOU
YOU

      2 3 /[1] 2 3ρ'HIMHER'
HIM
HIM
HER
HER
HER

      ¯1 2 ¯2 3 /[1] 2 3ρ'HIMHER'

HIM
HIM


HER
HER
HER

      R ← 3 2 4ρ'ME  YOU WE  THEYUS  THEM'
      R
ME
YOU

WE
THEY

US
THEM
```

```
      1 2 /[2] R
ME
YOU
YOU

WE
THEY
THEY

US
THEM
THEM
```

If L is entirely logical (containing only 0, 1, or both), then the derived function L/[A] is called Compress with Axis instead of Replicate with Axis.

The symbol ⌿ may be used instead of /.

Take: Z ← L ↑ R

R may be any array. L may be a simple scalar or vector of integers. If L is a scalar, then it will be treated as a one element vector. If R is a scalar, then it will be treated as a one element array with shape (⍴L)⍴1. After any scalar extensions, ⍴,L must be equal to ⍴⍴R.

Z is an array with the same rank as R, but with (possibly truncated or expanded) shape |L. If L[I] (an element of L) is positive, then L[I] sub-arrays are taken from the beginning of the Ith axis of R. If L[I] (an element of L) is negative, then |L[I] sub-arrays are taken from the end of the Ith axis of R.

If more elements are taken than exist on an axis in R, then the extra positions in Z are filled with the fill element (⊂∊⊃R).

Examples:

```
      3 ↑ 1 2 3 4 5
1 2 3

      ¯3 ↑ 1 2 3 4 5
3 4 5

      ¯4 ↑ 'NEPTUNE'
TUNE

      8 ↑ 1 2 3 4 5
1 2 3 4 5 0 0 0
```

```
      2 ¯3 ↑ 4 4ρι16
2 3 4
6 7 8

      2 ¯1 ¯3 ↑ 3 2 4ρ'ME  YOU WE  THEYUS  THEM'
OU

HEY

      Z ← 4 ↑ (1 2)(3 4 5)(7 8 9 10)
      Z
 1 2  3 4 5  7 8 9 10  0 0
      ρ¨Z
 2  3  4  2

      R ← 'ME' 'YOU'
      R
 ME YOU
      Z ← ¯5 ↑ R
      Z
          ME YOU
      ρ¨Z
 2  2  2  2  3
```

Take with Axis: Z ← L ↑[A] R

R may be any non-scalar array. L may be a simple scalar or vector of integers. A may be a simple scalar or vector integer selection of axes in R. A may not contain repetitions. Z is an array with the same rank as R, but with (possibly truncated) shape |L along axes A. The shape along axes not selected by A remains unchanged.

Conformability requires that (ρ,A)≤ρρR, and that ρ,L is ρ,A. If L is a scalar, then it will be treated as a 1-element vector.

Take with Axis is like the function Take, except that only the selected axes are affected.

Identity:

```
      L ↑ R  ↔  L ↑[ιρρR] R
```

Examples:

```
      (ι0) ↑[ι0] 1 2 3
1 2 3
```

```
      2 ↑[1] 4 4ρι16
1 2 3 4
5 6 7 8

      ¯6 ↑[2] 3 4ρι12
0 0  1  2  3  4
0 0  5  6  7  8
0 0  9 10 11 12

      ¯2 ↑[1] 3 2 4ρ'ME  YOU WE  THEYUS  THEM'
WE
THEY

US
THEM
```

More than one axis may be specified. If so, then ρ,*A* must be ρ,*L*.

Example:

```
      ¯2 3 ↑[1 3] 3 2 4ρ'ME  YOU WE  THEYUS  THEM'
WE
THE

US
THE
```

Multiple axes specified by *A* need not be in increasing order.

Example:

```
      3 ¯2 ↑[3 1] 3 2 4ρ'ME  YOU WE  THEYUS  THEM'
WE
THE

US
THE
```

Without: *Z* ← *L* ~ *R*

R may be any array. *L* may be any scalar or vector. *Z* is the vector of elements in *L* which do not occur in *R*. The length of *Z* is not greater than ρ,*L*.

Identity:

```
      L ~ R  ↔  (~L∊R)/L
```

Examples:

```
      'SHE' ~ 'S'
HE

      'MERCURY' ~ 'DUCKS'
MERRY

      'MERCURY' ~ 'MY' 'DUCKS'
MERCURY

      3 1 4 1 5 5 ~ 4 2 5 2 6
3 1 1
```

Note that the intersection of two vectors *L* and *R* (including any replications in *L*) may be obtained by the expression *L~L~R*.

Example:

```
      3 1 4 1 5 5 ~ 3 1 4 1 5 5 ~ 4 2 5 2 6
4 5 5
```

Note that the last part of the last example is the same as the previous example.

⎕*CT* is an implicit argument of Without.

PRIMITIVE DYADIC SELECTOR FUNCTIONS

The primitive dyadic selector functions are those that generate indices or a map of one array, dependent upon another array.

Deal: *Z ← L ? R*

R must be a simple scalar or one element vector containing a non-negative integer. *L* must be a simple scalar or one element vector containing a non-negative integer ≤ *R*. *Z* is an integer vector of length *L*, obtained by making *L* random selections without replacement from the set ⍳*R*.

Examples:

```
      3 ? 5
5 1 2

      3 ? 5
3 4 5
```

⎕*IO* and ⎕*RL* are implicit arguments of Deal. A side effect of Deal is to change the value of ⎕*RL* if *L*≠0.

Find: *Z ← L ⍷ R*

R may be any array. *L* may be any array. *Z* is a simple logical array of shape ρ*L*. An element of *Z* is 1 if the pattern *R* begins in the corresponding position of *L*. An element of *Z* is 0 otherwise.

If *R* has smaller rank than *L*, then *R* is treated as having shape ((*D*ρ1),ρ*R*)ρ*R*, where *D* is the difference in ranks. That is, the search is performed along the last ρρ*R* axes of *L*. If *R* has larger rank than *L*, then the pattern *R* cannot be found in *L*, and all elements of *Z* will be 0.

Examples:

```
      'A' ⍷ 'B'
0
```

```
      'A' ⍷ 'A'
1

      'ABABABA' ⍷ 'AB'
1 0 1 0 1 0 0

      'ABABABA' ⍷ 'A'
1 0 1 0 1 0 1

      1 2 3 4 5 2 3 4 5 ⍷ 2 3
0 1 0 0 0 1 0 0 0

      1 (2 3) (4 5) 2 3 4 5 ⍷ 2 3
0 0 0 1 0 0 0

      L ← 4 5⍴'ABCABA'
      L
ABCAB
AABCA
BAABC
ABAAB

      L ⍷ 'BA'
0 0 0 0 0
0 0 0 0 0
1 0 0 0 0
0 1 0 0 0

      L ⍷ 2 1⍴'BA'
0 1 0 0 1
0 0 1 0 0
1 0 0 1 0
0 0 0 0 0
```

⎕*CT* is an implicit argument of Find.

Find with Axis: *Z* ← *L* ⍷[*A*] *R*

R may be any array. *L* may be any array. *A* may be a simple scalar or vector integer selection of axes in *L*. *A* may not contain repetitions. The number of elements in *A* must be equal to the rank of *R*. *Z* is a simple logical array of shape ⍴*L*. An element of *Z* is 1 if the pattern *R* begins in the corresponding position of *L*, with the search performed along axes *A* of *L*.

Identity:

```
      L ⍷[(-⍴⍴R)↑⍳⍴⍴L] R   ←→   L ⍷ R
```

Find with Axis is like the function Find, except that axes other than the last in *L* may be specified.

Examples:

```
      L ← 4 5⍴'ABCABA'
      L
ABCAB
AABCA
BAABC
ABAAB

      L ⍷[1] 'BA'
0 1 0 0 1
0 0 1 0 0
1 0 0 1 0
0 0 0 0 0

      L ← 2 2 2⍴1 2 3
      L
1 2
3 1

2 3
1 2
      L ⍷[2] 2 1
0 1
0 0

1 0
0 0

      L ← 2 3 4⍴'ABCDEFGHIJKLMNOPQRSTUVWXY'
      L
ABCD
EFGH
IJKL

MNOP
QRST
UVWX

      L ⍷[1 2] 2 2⍴'FJRV'
0 0 0 0
0 1 0 0
0 0 0 0

0 0 0 0
0 0 0 0
0 0 0 0
```

```
       L ⍷[1 3] 2 2⍴'FGRS'
0 0 0 0
0 1 0 0
0 0 0 0

0 0 0 0
0 0 0 0
0 0 0 0
       L ⍷[2 3] 2 2⍴'FGJK'
0 0 0 0
0 1 0 0
0 0 0 0

0 0 0 0
0 0 0 0
0 0 0 0
```

Multiple axes specified by *A* need not be in increasing order.

Example:

```
       L ⍷[3 2] 2 2⍴'FJGK'
0 0 0 0
0 1 0 0
0 0 0 0

0 0 0 0
0 0 0 0
0 0 0 0
```

⎕*CT* is an implicit argument of Find with Axis.

Find Index: Z ← L ⍸ R

R may be any array. *L* may be any array. *Z* is an integer matrix containing the starting positions (in row major order) where pattern *R* begins in the pattern *L*. ¯1↑⍴*Z* is ⍴⍴*L*.

Find Index is defined in terms of the functions Find and Index Set:

```
       L ⍸ R   ↔   (,L ⍷ R)⌿ ,[⍳⍴⍴L] ⌷ ⍴L
```

for all valid non-scalar *L* and *R*.

Examples:

```
       ⍴ 'A' ⍸ 'B'
0 0
```

```
      ρ 'A' ⍷ 'A'
1 0

      'ABABABA' ⍷ 'AB'
1
3
5

      1 0 1 0 0 1 1 0 ⍷ 1
1 3 6 7

      1 2 3 4 5 2 3 4 5 ⍷ 2 3
2
6

      1 (2 3) (4 5) 2 3 4 5 ⍷ 2 3
4

      L ← 4 5ρ'ABCABA'
      L
ABCAB
AABCA
BAABC
ABAAB

      L ⍷ 'BA'
3 1
4 2

      L ⍷ 2 1ρ'BA'
1 2
1 5
2 3
3 1
3 4
```

⎕CT and ⎕IO are implicit arguments of Find Index.

Find Index with Axis: Z ← L ⍷[A] R

R may be any array. L may be any array. A may be a simple scalar or vector integer selection of axes in L. A may not contain repetitions. The number of elements in A must be equal to the rank of R. Z is an integer matrix containing the starting positions (in row major order) where pattern R begins in the pattern L, with the search performed along axes A of L. ¯1↑ρZ is ρρL.

Identity:

```
      L ⍸[(-⍴⍴R)↑⍳⍴⍴L] R  ↔  L ⍸ R
```

Find Index with Axis is like the function Find Index, except that axes other than the last in *L* may be specified.

Example:

```
      L ← 4 5⍴'ABCABA'
      L
ABCAB
AABCA
BAABC
ABAAB

      L ⍸[1 2] 2 1⍴'BA'
1 2
1 5
2 3
3 1
3 4

      L ← 2 2 2⍴1 2 3
      L
1 2
3 1

2 3
1 2

      L ⍸[2] 2 1
1 1 2
2 1 1

      L ← 2 3 4⍴'ABCDEFGHIJKLMNOPQRSTUVWXY'
      L
ABCD
EFGH
IJKL

MNOP
QRST
UVWX

      L ⍸[1 2] 2 2⍴'FJRV'
1 2 2

      L ⍸[1 3] 2 2⍴'FGRS'
1 2 2

      L ⍸[2 3] 2 2⍴'FGJK'
1 2 2
```

Multiple axes specified by *A* need not be in increasing order.

Example:

```
      L ⍸[3 2] 2 2⍴'FJGK'
1 2 2
```

⎕CT and ⎕IO are implicit arguments of Find Index with Axis.

Grade Down (Dyadic):   Z ← L ⍒ R

R may be any simple non-scalar character array. L may be any simple non-empty non-scalar character array, with no dimension exceding 256 in length. Z is a simple integer vector of shape 1↑⍴R, containing the permutation of ⍳1↑⍴R that puts the sub-arrays along the first axis of R in non-ascending order according to the collating sequence L.

Collation works by searching in L (in row major order) for each element of R, and then attaching a significance depending upon where it was first found. The significance depends upon both the location and the rank of L.

Any elements of R not found in L have collating significance as if they were found immediately past the end of L. Z leaves the order among elements of equal collating significance undisturbed.

Examples:

```
      'ABCDE' ⍒ 'DEAL'
4 2 1 3

      R ← 5 4⍴'DEALLEADDEADDEEDDALE'
      R
DEAL
LEAD
DEAD
DEED
DALE

      'ABCDE' ⍒ R
2 4 1 3 5
```

The last axis of L is the most significant for collating, and the first axis of L is the least significant. Thus, in the following example, differences in spelling have higher significance than differences in case:

```
      R ← 5 4⍴'dealDealdeadDeadDEED'
      R
deal
Deal
dead
Dead
DEED

      L ← 2 5⍴'abcdeABCDE'
      L
abcde
ABCDE

      Z ← L ⍒ R
      Z
5 2 1 4 3

      R[Z;]
DEED
Deal
deal
Dead
dead
```

Another application of a multi-dimensional left argument is illustrated by the default collating sequence, shown in Figure 4 on page 58.

⎕*IO* is an implicit argument of dyadic Grade Down.

Grade Up (Dyadic): *Z* ← *L* ⍋ *R*

R may be any simple non-scalar character array. *L* may be any simple non-empty non-scalar character array, with no dimension exceding 256 in length. *Z* is a simple integer vector of shape 1↑⍴*R*, containing the permutation of ⍳1↑⍴*R* that puts the sub-arrays along the first axis of *R* in non-descending order according to the collating sequence *L*. Any elements of *R* not found in *L* have equal collating significance as if they were found past the end of *L*.

Collation works by searching in *L* (in row major order) for each element of *R*, and then attaching a significance depending upon where it was first found. The significance depends upon both the location and the rank of *L*.

Any elements of *R* not found in *L* have collating significance as if they were found immediately past the end of *L*. *Z* leaves the order among elements of equal collating significance undisturbed.

Example:

```
      'ABCDE' ⍋ 'DEAL'
3 1 2 4

      R ← 5 4⍴'DEALLEADDEADDEEDDALE'
      R
DEAL
LEAD
DEAD
DEED
DALE

      'ABCDE' ⍋ R
5,3 1 4 2
```

The last axis of *L* is the most significant for collating, and the first axis of *L* is the least significant. Thus, in the following example, differences in spelling have higher significance than differences in case:

```
      R ← 5 4⍴'dealDealdeadDeadDEED'
      R
deal
Deal
dead
Dead
DEED

      L ← 2 5⍴'abcdeABCDE'
      L
abcde
ABCDE

      Z ← L ⍋ R
      Z
3 4 1 2 5

      R[Z;]
dead
Dead
deal
Deal
DEED
```

Another application of a multi-dimensional left argument is illustrated by the default collating sequence, shown in Figure 4 on page 58.

⎕*IO* is an implicit argument of dyadic Grade Up.

Index Of: Z ← L ⍳ R

R may be any array. L may be any vector. Z is an integer array with the same shape as R, describing where each element in R can first be found in L. If an element of R can not be found in L, then the corresponding element in Z will be ⎕IO+⊃⍴L.

Example:

```
      1 3 5 7 ⍳ 3 4 5
2 5 3
```

⎕CT and ⎕IO are implicit arguments of Index Of.

Member: Z ← L ∊ R

R may be any array. L may be any array. Z is a simple logical array of shape ⍴L. An element of Z is 1 if the corresponding element of L can be found anywhere in R.

Example:

```
      1 3 5 7 ∊ 3 4 5
0 1 1 0
```

⎕CT is an implicit argument of Member.

PRIMITIVE DYADIC MIXED FUNCTIONS

The primitive dyadic mixed numeric functions are those that are not pervasive, but apply to arrays *L* and *R*, and produce an array result *Z*, dependent upon the content of *L* and *R*.

Decode: *Z* ← *L* ⊥ *R*

R may be a simple real or complex numeric array. *L* may be a simple real or complex numeric array. *Z* is a simple real or complex numeric array of shape (¯1↓ρ*L*),1↓ρ*R*. Scalar arguments will be treated as one element vectors. If 1↑ρ*R* is 1, then the first axis of *R* will be extended to length ¯1↑ρ*L* before application of the function. If ¯1↑ρ*L* is 1, then the last axis of *L* will be extended to length 1↑ρ*R* before application of the function. Conformability requires that ¯1↑ρ*L* must be 1↑ρ*R*. Decode is defined in terms of the Inner Product +.×:

```
      L ⊥ R  ←→  ((ρL)↑⌽1,×\⌽1↓[ρρL]L) +.× R
```

for all valid non-scalar *L* and *R*.

Examples:

```
      2 ⊥ 1 0 1 0
10

      2 2 2 2 ⊥ 1 0 1 0
10

      2 2 2 2 ⊥ 0J1 0 1 0
2J8
```

If *L* is a scalar, then *L*⊥*R* is the value of the polynomial evaluated at *L*, with coefficients *R* (arranged in descending order of powers of *L*).

Encode: *Z* ← *L* ⊤ *R*

R may be a simple real or complex numeric array. *L* may be a simple real or complex numeric array. *Z* is a simple real or complex numeric array with shape (ρ*L*),ρ*R*.

Encode is defined in terms of the function Residue. For a vector *L* and a scalar *R*, *Z* may be determined by the following function:

```
    ∇ Z ←L ENCODE R;I
[1]    Z←0×L
[2]    I←ρL
[3]   GO:→(I=0)/0
[4]    Z[I]←L[I]|R
[5]    →(L[I]=0)/0
[6]    R←(R-Z[I])÷L[I]
[7]    I←I-1
[8]    →GO
    ∇
```

For arguments of other rank,

```
      Z ← ⊃[1] (⊂[1] L) ∘.(ENCODE¨) R
```

Examples:

```
      2 2 2 2 ⊤ 10
1 0 1 0

      2 0 2 ⊤ 13
0 6 1

      2 2 2 ⊤ ¯13
0 1 1

      2 2 2 2 ⊤ 2J8
0J1 0 1 0
```

The function Encode is the inverse of the function Decode for some vector arguments:

```
      L⊥L⊤R  ←→  (×/L)|R
```

⎕CT is an implicit argument of Encode.

Matrix Divide: Z ← L ⌹ R

L may be a simple real or complex scalar, vector, or matrix. R may be a simple real or complex scalar, vector, or matrix, subject to conformability with L, as described below. Z is a simple real or complex vector or matrix of shape (1↓ρR),1↓ρL minimizing the quantity +/,(L-R+.×Z)*2. The system variable Implicit Result (⎕IR) is set to the algebraic rank of R.

The definition assumes that L and R are matrices. If either L or R is a vector, then it is treated as a 1 column matrix. If either L or R is a scalar, then it is treated as a matrix with shape 1 1. After these extensions, L and R must have the same non-zero number of rows. If R has more columns than rows, then the system variable Matrix Divide Tolerance (⎕*MD*) must be non-zero.

Identity:

```
      I ⌹ R  ↔  ⌹ R
```

for matrix *R*, where *I* is an identity matrix of (square) shape 2ρ1↑ρ*R*.

If *R* is a non-singular square matrix, and *L* is a vector, then *Z* is the solution of the system of linear equations expressed conventionally as Rz=l. That is, *R*+.×*Z* is *L*.

Examples:

```
      1 4 ⌹ 2 2ρ1 0 0 2
1 2

      ⎕IR
2

      1 4 ⌹ 2 2ρ0J1 0 0 2
0J¯1 2
```

If *R* is a non-singular square matrix, and *L* is a matrix, then *Z* is the solution of the system of linear equations for each <u>column</u> of *L*. That is, *R*+.×*Z* is *L*.

Examples:

```
      (2 2ρ1 2 4 8) ⌹ 2 2ρ1 0 0 2
1 2
2 4

      (2 2ρ1 2 4 8) ⌹ 2 2ρ0J1 0 0 2
0J¯1 0J¯2
2    4
```

If the system variable Matrix Divide Tolerance (⎕*MD*) is 0, then *L*⌹*R* is executed only if:

1. *L* and *R* have the same number of rows

2. the columns of *R* are linearly independent

3. *R* does not have more columns than rows

If *R* is a vector, *F* is a numeric function, and *L*←*F R*, then *L*⌹*R*∘.*0,⍳*D* is the vector of the coefficients of the polynomial of degree *D* which best fits (in the least squares sense) the function *F* at points *R*. For example, to compute and evaluate successively closer polynomial approximations to the Gamma function:

```
      V ← 1 1.2 1.4 1.6 1.8 2
      L ← ! V
      L
1 1.101802491 1.242169345 1.429624559 1.676490788 2
      1.6 ⊥⌽ L⌹V∘.*0,⍳2
1.434010955
      1.6 ⊥⌽ L⌹V∘.*0,⍳3
1.428958487
      1.6 ⊥⌽ L⌹V∘.*0,⍳4
1.429580485
      1.6 ⊥⌽ L⌹V∘.*0,⍳5
1.429624559
```

where *X*⊥*P* evaluates a polynomial *P* at point *X* (see the function Decode).

Geometrically, if *R* is a matrix, and *L* is a vector, then *R*+.×*L*⌹*R* is a point in the space spanned by the column vectors of *R* which is closest to the point *L*. In other words, *R*+.×*L*⌹*R* is the projection of *L* on the space spanned by the columns of *R*.

If *R* is singular, or has more columns than rows, and if the system variable Matrix Divide Tolerance (⎕*MD*) is non-zero, then ⎕*MD* is taken to be a fuzz on the algebraic rank determination of *R*. The behavior of the system variables ⎕*MD* and ⎕*IR* in such a case follows from the identity:

```
      L ⌹ R   ←→   (⌹R)+.×L
```

(see Matrix Inverse on page 65)

Example:

```
      R ← 3 3⍴1  0  0  1  0  0  0  0  2
      R
1 0 0
1 0 0
0 0 2
      1 2 4 ⌹ R
DOMAIN ERROR
      1 2 4⌹R
      ∧    ∧

      ⎕MD ← 1E¯13
      1 2 4 ⌹ R
1.5 0 2

      ⎕IR
2
```

⎕*IR* is an implicit result of Matrix Divide. ⎕*MD* is an implicit argument of Matrix Divide. ⎕*MD* is not related to ⎕*CT*.

The Matrix Divide and Matrix Inverse functions use the "Lawson and Hansen Algorithm", which is an extension of the "Golub and

Businger Algorithm", to handle undetermined cases. If ⎕*MD* is 0, then the test for singularity uses a fixed implicit fuzz of 1*E*¯15. For statistical problems with experimental data, the value of ⎕*MD* should reflect the relative accuracy of the data.

PRIMITIVE DYADIC TRANSFORMATION FUNCTIONS

The primitive dyadic transformation functions are those that are not pervasive, but apply to arbitrary arrays *L* and *R*, and produce an array result *Z* with data type independent of that of their arguments.

Match: *Z* ← *L* ≡ *R*

R may be any array. *L* may be any array. *Z* is a simple logical scalar (either 0 or 1). If *L* is identical to *R*, then *Z* is 1. If *L* is not identical to *R*, then *Z* is 0.

Non-empty arrays are identical if they have the same structure and the same values in all corresponding locations. Empty arrays are identical if they have the same shape and the same prototype (disclosed nested structure).

Examples:

```
      2 ≡ 2
1

      2 ≡ 3
0

      2 ≡ ,2
0

      '' ≡ ⍳0
0

      (0⍴⊂0 0) ≡ 0⍴⊂0 0 0
0

      'ME' ≡ 'ME '
0

      'ME' ≡ ⊂'ME'
0

      'ME' 'YOU' ≡ 'ME' 'YOU'
1

      2 3 4 ≡ 1+1 2 3
1
```

```
      2 3 4 ≡ 2 3 4.000000000000001
1
```

⎕*CT* is an implicit argument of Match.

Format (Dyadic):   *Z* ← *L* ⍕ *R*

R may be an array of any rank. *R* may not have any items which are complex scalars. If *R* is non-simple, then the ranks of its items must all be less than two. *L* may be a simple scalar or vector. *Z* is a character array displaying the array *R* according to the specification *L*. *Z* has rank 1⌈⍴⍴*R*, and ¯1↓⍴*Z* is ¯1↓⍴*R*. ⎕*FC* (described on page 210) is an implicit argument of dyadic Format.

Conformability requires that if *L* is numeric and has more than two elements, then ⍴,*L* must be 2×¯1↑⍴*R*. If *L* is numeric and has two elements, then it will be extended to 2×¯1↑⍴*R* elements.

The specifications *L* may have one of three forms:

1. two numbers for each column of *R*

2. a single integer (the same as (0,*L*)⍕*R*)

3. a character vector (Picture Format)

If *L* has two numbers for each column of *R*, and the first is positive, then it specifies the total column width. In such a case, the second number:

> If positive or zero, specifies the number of digits displayed after the decimal point for numeric scalar items in the corresponding column of *R*.
>
> Examples:
>
> ```
>       7 2 ⍕ 0 1.1 21.12 321.123
>     .00   1.10   21.12 321.12
>
>       4 0 ⍕ 0 1.1 21.12 321.123
>    0   1   21 321
> ```
>
> If negative, specifies the number of digits displayed in the mantissa of a floating point representation for numeric scalar items in the corresponding column of *R*.

Example:

```
      7 ¯3 ⍕ 1 21 321 4321
  1.00E0 2.10E1 3.21E2 4.32E3
```

The digits specification is ignored for non-numeric or non-scalar items in R. If they fit, then they will be right-adjusted in their columns. If they don't fit, and ⎕FC[6] is not '0', then the corresponding fields of Z will be filled with ⎕FC[4].

Examples:

```
      3 0 5 1 5 2 ⍕ 3 3⍴'ONE' 'TWO' 'THREE' 1 20 3 4 5 0.7
ONE  TWO THREE
  1 20.0  3.00
  4  5.0  0.70

      ⎕FC[4] ← '?'

      3 0 5 1 3 2 ⍕ 3 3⍴'ONE' 'TWO' 'THREE' 1 20 3 4 5 0.7
ONE  TWO ???
  1 20.0 ???
  4  5.0 .70
```

If the first number of a pair in L is zero, then it specifies a floating column width determined by the contents in the corresponding column of R.

Example:

```
      0 2 ⍕ 0 1.1 21.12 321.123
 .00 1.10 21.12 321.12
```

If L is a single integer, then this is the same as (0,L)⍕R.

Example:

```
      2 ⍕ 0 1.1 21.12 321.123
 .00 1.10 21.12 321.12
```

If L is a character vector, then it specifies ¯1↑⍴Z and the resulting pattern of Z. R must be numeric, and the individual characters in L control the display of columns in R. This is called Picture Format. ⎕FC is an implicit argument of Picture Format.

In Picture Format, digits (numeric characters) in the pattern L are control characters, and show where digits may appear in the result Z. Non-digits (non-numeric characters) in the pattern L are called decorators. Decorators may be either embedded or conventional.

An embedded decorator may be either fixed in position, or it may be controlled. A controlled decorator may print or not, or it may float next to the number in the result, depending upon which control digits are used, and what number is being formatted.

The dot and the comma are conventional decorators, which indicate decimal points and commas in the result by known conventions.

Control characters are:

```
    0  pad zeros to this position
    1  float decorator if negative
    2  float decorator if non-negative
    3  float decorator
    4  do not float nearest decorator
    5  normal digit
    6  field ends at right of non-control character
    7  exponential symbol at right of non-control character
    8  fill with ⎕FC[3] when otherwise blank
    9  pad zeros to this position if non-zero
    .  decimal point
    ,  controlled comma

       all other characters are decorators
```

A field in a pattern is a sequence of characters containing at least one digit, and bounded by either blanks or special field boundary markers (like the digit 6). If a sequence of characters does not contain a digit, then it is considered a decoration.

The normal digit to use in the pattern is 5. A field of only 5's will suppress leading and trailing zeros. If there is only one field, then it is used for every column of numbers in *R*:

```
      Z ← ' 555.55' ⍕ 1 0 10.1 100
      Z
   1           10.1   100
      ⍴Z
28
```

If there is more than one field, then there must be one for every column of numbers in *R*:

```
      Z ← ' 5 5.5 5.55' ⍕ 1.12 2.12 3.12
      Z
 1 2.1 3.12
      ⍴Z
11
```

A 0 can be used in the field to pad zeros to a particular point:

```
      Z ← ' 005 5.50 5.550' ⍕ 1.12 2.12 3.12
      Z
 001 2.12 3.120
      ⍴Z
15
```

Embedded decorators may be included:

```
      Z ← 'HERE 5 5.5 ;THERE: 5.55' ⍕ 1.12 2.12 3.12
      Z
HERE: 1 2.1 ;THERE: 3.12
      ⍴Z
24
```

A single field may have embedded decorators:

```
      Z ← '05/05/05' ⍕ 70481
      Z
07/04/81
      ⍴Z
8
```

A 1 can be used in the field to float a decorator in against a number for negative values only:

```
      Z ← ' -551.50' ⍕ ¯1 0 10 ¯100
      Z
   -1.00     .00   10.00 -100.00
      ⍴Z
32
```

A floating decorator may be on both sides of a number:

```
      Z ← '(551.50)' ⍕ ¯1 0 10 ¯100
      Z
  (1.00)    .00   10.00 (100.00)
      ⍴Z
32
```

A 2 can be used in the field to float a decorator in against a number for non-negative values only:

```
      Z ← ' +552.50' ⍕ ¯1 0 10 ¯100
      Z
    1.00    +.00  +10.00  100.00
      ⍴Z
32
```

A 3 can be used in the field to float a decorator in against a number for all values:

```
      Z ← ' $553.50' ⍕ 1 0 10 100
      Z
   $1.00    $.00  $10.00 $100.00
      ⍴Z
32
```

A 4 can be used with a 1, 2, or 3 in the field to mix non-floating and floating decorators. It blocks the floating effect of a 1, 2, or a 3 on its side of the decimal point.

```
      Z ← ' -551.45*' ⍕ ¯1 0 10.1 ¯100
      Z
   -1   *        *   10.1 * -100   *
      ⍴Z
36
```

A 6 can be used to end a field which is otherwise continued. It indicates that not only a blank, but any non-control character ends a field.

```
      Z ← '06/06/06' ⍕ 7 4 81
      Z
07/04/81
      ⍴Z
8
```

A 7 can be used to indicate a double field for scaled formatting. The next decorator to the right of a 7 replaces the *E* in scaled form:

```
      Z ← '1.70*00' ⍕ 12345
      Z
1.23*04
      ⍴Z
7
```

An 8 can be used in the field to have otherwise blank positions in the result filled with ⎕*FC*[3]:

```
      Z ← ' 8555.50' ⍕ 1 0 10 100
      Z
 ***1.00 ****.00 **10.00 *100.00
      ⍴Z
32
```

A 9 can be used in the field to pad zeros to a particular point only for non-zero numbers:

```
      Z ← ' 555.59' ⍕ 1 0 100
      Z
   1.00          100.00
      ⍴Z
21
```

If ⎕FC[4] is not a 0, then it is used to fill a field that would otherwise be an error because the number is too large.

```
      Z ← ' 555.59' ⍕ 1 1000 100
DOMAIN ERROR
      Z←' 555.59'⍕1 1000 100
        ∧         ∧

      ⎕FC[4] ← '?'

      Z ← ' 555.59' ⍕ 1 1000 100
      Z
   1.00 ?????? 100.00
      ⍴Z
21
```

For more examples, refer to the Format Control system variable (⎕FC), on page 210.

Each of the primitive miscellaneous functions has one or more of the following properties:

1. It produces no explicit result.

2. It takes no explicit argument.

3. It does not follow the syntax of a monadic or dyadic function (*F R* or *L F R*).

Primitive miscellaneous functions are not in the function domain of operators.

Abort: →

This is a special case of the branch statement which means to clear the most recently suspended statement and all its pendent statements from the state indicator. The abort expression has no explicit result, and it takes no argument. The abort expression is not in the function domain of operators.

If → is executed in a defined function or operator, then it (and any pendent function or operator) is aborted.

Example:

```
      ∇ F
[1]    1
[2]    G
[3]    3
      ∇

      ∇ G
[1]    10
[2]    →
[3]    30
      ∇

       F
1
10
```

Bracket Indexing: *Z* ← *R* [*L*]

If there are no semicolons between the brackets, then *R* may be any vector. *L* may be a (possibly nested) array of integers not less than ⎕*IO*. Selected element(s) in *R* with indices *L* are referenced, forming an array *Z*. Bracket Indexing does not follow the syntax of a dyadic function. Bracket Indexing is not in the function domain of operators.

Example:

```
      R ← 1 2 3 4 5
      R[2]
2
```

If *R* is a vector, then *R*[*I*] has the same shape as *I*, which may have any rank.

Example:

```
      R ← 1 2 3 4 5
      R[2 3⍴1 2 3 2 3 4]
1 2 3
2 3 4
```

If *L* is nested, then *Z* is nested with the same structure.

Example:

```
      R ← 1 2 3 4 5
      Z ← R[⊂2 3⍴1 2 3 2 3 4]
      Z
 1 2 3
 2 3 4
      ⍴⍴Z
0
      ⍴⊃Z
2 3
```

If *R* is a matrix, then two arrays of indices may be given, separated by a semicolon (;). They reference the rows and the columns, respectively.

Example:

```
      R ← 3 3⍴1 2 3 4 5 6 7 8 9
      R[1;3]
3
```

If *R* is a matrix, then *R*[*I*;*J*] has the shape (⍴*I*),⍴*J*.

Example:

```
      R ← 3 3⍴1 2 3 4 5 6 7 8 9
      R[1 2;2 3⍴1 2 2 3 3 2]
1 2 2
3 3 2

3 4 4
5 5 4
```

Index arrays may be elided to indicate all indices for the corresponding axis.

Examples:

```
      R ← 3 3⍴1 2 3 4 5 6 7 8 9

      R[1;]
1 2 3

      R[2 1;]
4 5 6
1 2 3

      R[;1]
1 4 7

      R ← 2 3⍴'YOUMAY'

      R[1;]
YOU

      R[;1]
YM
```

Arrays of any non-zero rank *D* may be referenced by Bracket Indexing if there are *D*-1 semicolons between the brackets, and *D* optional arrays of indices.

Example:

```
      R ← 3 2 4⍴'ME  YOU WE  THEYUS  THEM'
      R
ME
YOU

WE
THEY

US
THEM

      R[1;2;]
YOU

      R[1;2;3]
U
```

⎕IO is an implicit argument of Bracket Indexing.

Example:

```
      ⎕IO ← 0
      R ← 1 2 3 4 5
      R[0 2]
1 3
```

Branch: → R

R may be a scalar or vector, which, if not empty, has an simple integer scalar as its first item. The branch expression has no explicit result. The branch expression is not in the function domain of operators. It is used to modify the normal sequential flow of control in a defined function or operator, or to resume execution after a statement has been interrupted.

There are four distinct uses for the branch expression, depending upon whether or not the argument is empty, and whether or not the statement is entered in immediate execution: They are shown in Figure 7 on page 149.

	Entered in a Defined Function or Operator	Entered in Immediate Execution
→*LINE*	Continue with a specific line	Restart at the beginning of a line
→⍳0	Continue with the next line	Resume in the middle of a line

Figure 7. Uses of the Branch Expression

If *R* is non-empty, and the branch is executed in a defined function or operator, then the first element of *R* specifies the number of the line to be executed next, if it exists. Other elements of *R* after the first are ignored.

Example:

```
     ∇ F
[1]   1
[2]   G
[3]   3
     ∇

     ∇ G
[1]   10
[2]   →4
[3]   30
[4]   40
     ∇

       F
1
10
40
3
```

If the line number doesn't exist, then execution of the function or operator is terminated. (Line 0 does not exist for branching purposes.)

Example:

```
     ∇ F
[1]   1
[2]   G
[3]   3
     ∇
```

```
      ∇ G
[1]    10
[2]    →0
[3]    30
[4]    40
      ∇

         F
1
10
3
```

The argument *R* of a Branch statement may (in fact, should) be a label. A label is a name which is followed by a colon (:) at the beginning of a statement. It is effectively a local constant which is given a value when execution of the defined function or operator is begun. A label does not affect the execution of the statement on which it appears. (See also "System Labels" on page 227.)

Example:

```
      ∇ G
[1]    10
[2]    →FOUR
[3]    30
[4] FOUR: 40
      ∇

         G
10
40
```

In this example, *FOUR* is a label.

If *R* is empty, and the branch is executed in a defined function or operator, then no branch takes place, and execution continues with the next line in sequence.

Example:

```
      ∇ G
[1]    10
[2]    →⍳0
[3]    30
[4]    40
      ∇

         G
10
30
40
```

In this manner, the branch statement may be conditional.

Example:

```
      ∇ G
[1]   10
[2]   →MAYBE/FOUR
[3]   30
[4]   FOUR: 40
      ∇

        MAYBE←0
        G
10
30
40

        MAYBE←1
        G
10
40
```

If the branch statement is conditional, an iterative procedure may be performed.

Example:

```
      ∇ Z←FACTORIAL R
[1]    Z←R⌈1
[2]  LOOP: R←R-1
[3]    →(R≤1)/0
[4]    Z←Z×R
[5]    →LOOP
      ∇
```

Similarly, the branch statement may have multiple paths.

Example:

```
      ∇ G
[1]    10
[2]    →(THREE,FIVE,SEVEN)[WHICH]
[3]  THREE: 30
[4]    →0
[5]  FIVE: 50
[6]    →0
[7]  SEVEN: 70
      ∇

        WHICH←1
        G
10
30
```

```
      WHICH←2
      G
10
50

      WHICH←3
      G
10
70
```

If *R* is non-empty, and if the branch is executed in immediate execution, then the branch is taken as a request to restart execution of the last suspended defined function or operator at the specified line, if the line exists. If it doesn't exist, then execution of the function or operator is terminated.

This action only restarts execution of a defined function or operator. If there are suspended lines of immediate execution in the state indicator above the function or operator, then they are lost. If there is no suspended function or operator, then the branch expression does nothing.

Examples:

```
    ∇ G
[1]  10
[2]  ÷0
[3]  30
    ∇

      G
10
DOMAIN ERROR
G[2] ÷0
     ∧∧
      →3
30


    ∇ F
[1]  1
[2]  G
[3]  3
    ∇

    ∇ G
[1]  10
[2]  ÷0
[3]  30
    ∇
```

```
      F
1
10
DOMAIN ERROR
G[2] ÷0
     ∧∧
      )SI
G[2]
F[2]
*
      →0
3
```

If *R* is empty, and if the branch is executed in immediate execution, then the branch is taken as a request to resume execution at the current position in the most recently suspended line. The most recently suspended line may be in a defined function or operator, or it may have been entered in previous immediate execution.

Resuming a line may cause re-evaluation of an expression which gave an error. If the error was not a *SYNTAX ERROR* or a *VALUE ERROR*, then the system variable ⎕*R* and perhaps ⎕*L* will be available for inspection or re-assignment. In this case the new values of system variables ⎕*L* and ⎕*R* are used in the re-evaluation if they have been provided. For more details, refer to the description of the system variable Right Argument (⎕*R*) on page 220.

Example:

```
    ∇ G
[1]  10
[2]  ÷0
[3]  30
    ∇

      G
10
DOMAIN ERROR
G[2] ÷0
     ∧∧
      ⎕R
0
      ⎕R←2
      →⍳0
0.5
30
```

Example:

```
      2×V+1
VALUE ERROR
      2×V+1
        ∧

      V ← 3
      →ι0
7
```

Example:

```
      ⎕IO ← 2.71828

      2×ι4
⎕IO ERROR
      2×ι4
        ∧

      ⎕IO ← 1

      →ι0
2 4 6 8
```

Refer also to the system variable Line Counter (⎕*LC*). The statement →⎕*LC* is a convenient way to restart anew (rather than resume from the current position) execution of a suspended line of a defined function or operator.

The APL2 primitive operators take one or two operands and produce a derived function. Some operators take one operand (which must be a function). They are called monadic operators. Some operators take two operands. They are called dyadic operators. The second operand of a dyadic operator is not optional.

The left operand of an operator may be a primitive function, a system function, a defined function, or a derived function. The right operand of an operator may be an array, a primitive function, a system function, a defined function, or a derived function. The resulting derived function is ambi-valent, and may have both monadic and dyadic definitions.

Class	Name	Producing Monadic	Pg	Producing Dyadic	Pg
Monadic	Each	*F*¨ *R*	156	*L F*¨ *R*	157
	Reduce	*F*/ *R*	160	*L F*/ *R*	158
	Reduce	*F*⌿ *R*	160	*L F*⌿ *R*	158
	Scan	*F*\ *R*	165		
	Scan	*F*⍀ *R*	165		
	Reduce w Axis	*F*/[*A*] *R*	165	*L F*/[*A*] *R*	159
	Scan w Axis	*F*\[*A*] *R*	167		
	Bracket Axis	*F*[;] *R*	168	*L F*[;;] *R*	172
Dyadic	Inner Product			*L F*.*G R*	178
	Outer Product			*L* ∘.*G R*	176

Notes:
F and *G* are function operands of an operator.
A is a simple scalar or vector axis specification.
L and *R* are array arguments of a derived function.

Figure 8. Primitive Operators

PRIMITIVE MONADIC OPERATORS

The primitive monadic operators take a left operand (which must be a function) and produce a monadic or dyadic derived function. The monadic operators are presented by defining the derived functions they produce.

Each (producing Monadic): Z ← F¨ R

F may be any monadic function. R may be any array whose items are appropriate to the function F. Z is an array of shape ⍴R formed by applying F to each item of R.

If R is not empty, then (,Z)[I] is ⊂F ⊃(,R)[I] for every simple scalar I for which (,R)[I] is defined.

If R is empty and F is primitive, then the argument presented to F is ∊⊃R, which is the same as ⊃R. If R is empty and F is defined, then F must contain the system label □FL (see "System Labels" on page 227).

Examples:

```
      ⍴¨ 'ME' 'YOU' 'WE'
 2  3  2

      ⍳¨ 0 1 2 3
    1  1 2  1 2 3

      ⍳¨ 1 2 3 4
 1  1 2  1 2 3  1 2 3 4

      Z ← ⍕¨ 0⍴0
      ⍴Z
0
      ⍴⊃Z
1
```

The monadic derived function F¨ is a scalar function, but it is not a pervasive one unless F is pervasive. If applied to a pervasive function, the Each operator has no effect.

Each (producing Dyadic): Z ← L F¨ R

F may be any dyadic function. L and R must be identically shaped arrays (after possible extensions) whose corresponding items are appropriate to the function F. Z is an array of shape ρL (or ρR) formed by applying F between each corresponding pair of items in L and R.

The following extensions will be performed if applicable before application of the derived function:

1. If L is a scalar or a 1 element vector, then it is reshaped to shape ρR.

2. If R is a scalar or a 1 element vector, then it is reshaped to shape ρL.

If L and R are not empty after any scalar extensions, then (,Z)[I] is ⊂(⊃(,L)[I]) F ⊃(,R)[I] for every simple scalar I for which (,L)[I] and (,R)[I] are defined.

If either L or R is empty and F is primitive, then the arguments presented to F are ∊⊃L and ∊⊃R. If either L or R is empty and F is defined, then F must contain the system label ⎕FL (see "System Labels" on page 227).

Examples:

```
      4 6 1 ρ¨ 'ME' 'YOU' 'WE'
 MEME YOUYOU W

      6 ρ¨ 'ME' 'YOU' 'WE'
 MEMEME YOUYOU WEWEWE

      (1ρ6) ρ¨ 'ME' 'YOU' 'WE'
 MEMEME YOUYOU WEWEWE

      2 4 ρ¨ 'X'
 XX XXXX

      Z ← 4↓¨ 0ρ⊂0 0 0
      ρZ
0
      ρ⊃Z
3
```

The dyadic derived function F¨ is a scalar function, but not necessarily a pervasive one. If applied to a pervasive function, the Each operator has no effect.

N-Wise Reduce: Z ← L F/ R

F may be any dyadic function which produces a result. R must be an array whose sub-arrays along the last axis are appropriate to the derived function F/. L must be a simple scalar or one element vector integer such that (|L)≤¯1↑ρR. If F is a scalar (or pervasive) function, then Z is an array with the same rank as R, but with shape (¯1↓ρR),1+(¯1↑ρR)-|L.

Z is formed by applying the derived function F/ to contiguous sub-arrays of width L along the last axis of R. If L is negative, the specified sub-arrays of R will be reversed along the last axis before the applications of the derived function F/.

If L is 0 then the result contains identity arrays for F with respect to R. If L is 0 and F is defined, then F must contain the system label ⎕ID (see "System Labels" on page 227). If L is 0, then F may not be derived except when directly produced by a defined operator containing the system label ⎕ID.

Identity if F is a scalar or pervasive function:

```
      ¯1↑ρZ  ←→  1+(¯1↑ρR)-|L
```

for all valid L and R.

Examples:

```
      5 +/ 1 4 9 16 25
55

      4 +/ 1 4 9 16 25
30 54

      3 +/ 1 4 9 16 25
14 29 50

      2 +/ 1 4 9 16 25
5 13 25 41

      1 +/ 1 4 9 16 25
1 4 9 16 25

      0 +/ 1 4 9 16 25
0 0 0 0 0 0

      ¯2 -/ 1 4 9 16 25
3 5 7 9

      2 =/ 'MERRY'
0 0 1 0
```

```
      3 ,/ 'ABCDEF'
ABCBCDCDEDEF

      3 ,/ 1 2 3 4 5 6 7
1 2 3 2 3 4 3 4 5 4 5 6 5 6 7

      3 ,¨/ 'ABCDEF'
 ABC BCD CDE DEF

      3 ,¨/ 1 2 3 4 5 6 7
 1 2 3  2 3 4  3 4 5  4 5 6  5 6 7

      ¯3 ,¨/ 1 2 3 4 5 6 7
 3 2 1  4 3 2  5 4 3  6 5 4  7 6 5
```

The symbol ⌿ may be used instead of / to indicate the first axis of *R* rather than the last.

Example:

```
      2 +⌿ 3 4⍴⍳12
 6  8 10 12
14 16 18 20
```

N-Wise Reduce with Axis: *Z ← L F/[A] R*

F may be any dyadic function which produces a result. *R* must be an array whose sub-arrays along axis *A* are appropriate to the derived function *F/*. *L* must be a simple scalar or one element vector integer such that (|*L*)≤(⍴*R*)[*A*]. *A* may be a simple scalar or a one element vector containing an integer axis in *R*. If *F* is a scalar (or pervasive) function, then *Z* is an array with the same rank as *R*, but with shape 1+(¯1↑⍴*R*)-|L along axis *A*.

Z is formed by applying the derived function *F/*[*A*] to contiguous sub-arrays of width *L* along axis of *A* of *R*. If *L* is negative, the specified sub-arrays of *R* will be reversed along the last axis before the applications of the derived function *F/*.

If *L* is 0 then the result contains identity arrays for *F* with respect to *R*. If *L* is 0 and *F* is defined, then *F* must contain the system label ⎕*ID* (see "System Labels" on page 227). If *L* is 0, then *F* may not be derived except when directly produced by a defined operator containing the system label ⎕*ID*.

This operator is like N-Wise Reduce, except that an existing axis in *R* other than the default may be specified.

Example:

```
      2 +/[2] 1 3 4⍴⍳12
 6  8 10 12
14 16 18 20
```

The symbol ⌿ may be used instead of /.

Reduce: Z ← F/ R

F may be any dyadic function which produces a result. *R* must be an array whose sub-arrays along the last axis are appropriate to the function *F*. *Z* is an array formed by applying *F* between sub-arrays along the last axis of *R*. The arguments presented to *F*, if any, have rank one less than the rank of *R*. If *F* is a scalar (or pervasive) function, then *Z* has shape ¯1↓⍴*R*.

If *R* is a scalar, then *Z* is *R*. If 1=¯1↑⍴*R*, then *Z* is (¯1↓⍴*R*)⍴*R*. If 1<¯1↑⍴*R*, then the definition is recursive:

```
     F/ R  ←→  (1 ⌷[⍴⍴R] R) F F/1↓[⍴⍴R]R.
```

If 0=¯1↑⍴*R*, then the empty array *R* is presented to the monadic identity function. The identity function is the first of:

1. The right identity function *RI* for *F*, such that:

   ```
   A  ←→  A F RI ((⍴A),0)⍴A
   ```

2. The left identity function *LI* for *F*, such that:

   ```
   A  ←→  (LI ((⍴A),0)⍴A) F A
   ```

If 0=¯1↑⍴*R*, and neither a right nor a left identity function exists for *F*, then *F*/*R* is a *DOMAIN ERROR*.

Figure 9 on page 161 shows the identity elements for the primitive dyadic pervasive functions in the reduction of empty numeric arrays. If *F* is pervasive, and *I* is its identity element, then its identity function is (¯1↓⍴*R*)⍴*I*+⊂⊃*R*. This identity holds only for uniform arrays.

Figure 10 on page 162 shows the identity functions for the primitive dyadic non-pervasive functions in the reduction of empty arrays. If ¯1↑⍴*R* is 0 and *F* is defined, then *F* must contain the system label ⎕*ID* (see "System Labels" on page 227). If ¯1↑⍴*R* is 0, then *F* may not be derived except when directly produced by a defined operator containing the system label ⎕*ID*.

Function		Identity Element	Left/ Right	Identity Restriction
Add	+	0	L R	
Subtract	-	0	R	
Multiply	×	1	L R	
Divide	÷	1	R	
Residue	|	0	L	
Minimum	⌊	*M*	L R	
Maximum	⌈	-*M*	L R	
Power	*	1	R	
Logarithm	⍟		none	
Circular	○		none	
Binomial	!	1	L	
And	∧	1	L R	∧/,*A*∊0 1
Or	∨	0	L R	∧/,*A*∊0 1
Less	<	0	L	∧/,*A*∊0 1
Not Greater	≤	1	L	∧/,*A*∊0 1
Equal	=	1	L R	∧/,*A*∊0 1
Not Less	≥	1	R	∧/,*A*∊0 1
Greater	>	0	R	∧/,*A*∊0 1
Not Equal	≠	0	L R	∧/,*A*∊0 1
Nand	⍲		none	
Nor	⍱		none	

Note:
A is the array satisfying the identity.
M is 7.2370055773322621*E*75.

Figure 9. Identity Elements for Dyadic Pervasive Functions

Examples:

```
      +/ 5
5

      +/ 3⍴5
15

      +/ 2⍴5
10

      +/ 1⍴5
10

      +/ 0⍴5
0
```

Function	F	Identity Function (↔ *F*/*R*)	Left/ Right	Identity Restriction
Reshape	⍴	*S*	L	
Catenate	,	((¯1↓*S*),0)⍴*R*	L R	1≤⍴⍴*A*
Rotate	⌽	(¯1↓*S*)⍴0	L	
Rotate	⊖	(1↓*S*)⍴0	L	
Transpose	⍉	⍳0⌈¯1+⍴⍴*R*	L	
Pick	⊃	⍳0	L	
Drop	↓	(0⌈¯1+⍴⍴*R*)⍴0	L	
Take	↑	*S*	L	
Replicate	/	(¯1↑*S*)⍴1	L	1≤⍴⍴*A*
Replicate	⌿	(1↑*S*)⍴1	L	1≤⍴⍴*A*
Expand	\	(¯1↑*S*)⍴1	L	1≤⍴⍴*A*
Expand	⍀	(1↑*S*)⍴1	L	1≤⍴⍴*A*
Index	⌷	⌷*S*	L	
Without	~	0⍴*R*	R	1=⍴⍴*A*
Index of	⍳	⍳*M* {note[1]}	L	∧/,*A*∊⍳*M*
Member	∊	*S*⍴1	R	∧/,*A*∊0 1
Grade Up	⍋		none	
Grade Down	⍒		none	
Deal	?		none	
Find	⍷	1	R	∧/,*A*∊0 1
Find Index	⍸		none	
Encode	⊤	*M*	L	
Decode	⊥		none	
Mat. Divide	⌹	*V*∘.=*V*←⍳1↑⍴*R*	R	1≤⍴⍴*A*
Format	⍕		none	
Match	≡		none	

Notes:

R is the empty array being reduced (⍴*R* ↔ *S*,0).
A is the array satisfying the identity.
M is 7.237005577332262*E*75.
[1] ⍳*M* is a *DOMAIN ERROR*.

Figure 10. Identity Functions for Primitive Dyadic Non-Pervasive Functions

```
      R ← 3 4⍴1 2 3 4 5 6 7 8 9 10 11 12
      R
1  2  3  4
5  6  7  8
9 10 11 12
      +/ R
10 26 42
```

```
      +/ 4 3ρ5
15 15 15 15

      +/ 4 2ρ5
10 10 10 10

      +/ 4 1ρ5
5 5 5 5

      +/ 4 0ρ5
0 0 0 0
      +/ 3ρ(1 2 3 4)(10 20 30 40)(100 200 300 400)
111 222 333 444

      +/ 2ρ(1 2 3 4)(10 20 30 40)(100 200 300 400)
11 22 33 44

      +/ 1ρ(1 2 3 4)(10 20 30 40)(100 200 300 400)
1 2 3 4

      +/ 0ρ(1 2 3 4)(10 20 30 40)(100 200 300 400)
0 0 0 0

      +/¨ (1 2 3 4)(10 20 30 40)(100 200 300 400)
10 100 1000

      Z ← ρ/ 2ρ5
      ρZ
5
      Z
5 5 5 5 5

      Z ← ρ/ 1ρ5
      ρρZ
0
      Z
5

      Z ← ρ/ 0ρ5
      ρZ
0
      Z
      Z ← ρ/ 2 2ρ5
      ρZ
 5 5
      Z
 5 5 5 5 5
 5 5 5 5 5
 5 5 5 5 5
 5 5 5 5 5
 5 5 5 5 5

      Z ← ρ/ 2 1ρ5
      ρZ
 2
      Z
 5 5

      Z ← ρ/ 2 0ρ5
      ρZ
 1
      Z
 2

      ρ/ 2 2ρ2 4 3 5
 4 5 4
 5 4 5

      ,/ 2 3ρ1 2 3 4 5 6
 1 4 2 5 3 6

      ,¨/ 2 3ρ1 2 3 4 5 6
   1 2 3  4 5 6

      ≡/ 'ME' 'ME'
 1

      ≡/ 'ME' 'YOU'
 0
```

The symbol ⌿ may be used instead of / to indicate the first axis of *R* rather than the last.

Example:

```
      R ← 3 4ρ1 2 3 4 5 6 7 8 9 10 11 12
      R
1  2  3  4
5  6  7  8
9 10 11 12
      +⌿ R
15 18 21 24
```

Reduce with Axis: Z ← F/[A] R

F may be any dyadic function which produces a result. A may be a simple scalar or a one element vector containing an integer axis in R. R must be an array whose sub-arrays along axis A are appropriate to the function F. Z is an array formed by applying F between sub-arrays along axis A of R. The arguments presented to F, if any, have rank one less than the rank of R. If F is a scalar (or pervasive) function, then Z has shape (A≠⍳⍴⍴R)/⍴R.

This operator is like Reduce except that an existing axis in R other than the default may be specified by A.

Identity:

```
      F/[A] R   ←→   F/ (⍋((⍳⍴⍴R)~A),A)⍉R
```

Example:

```
      +/[2] 2 3 4⍴⍳24
15 18 21 24
51 54 57 60
```

The symbol ⌿ may be used instead of /.

Scan: Z ← F\ R

F may be any dyadic function which produces a result. R must be an array whose sub-arrays along the last axis are appropriate to the derived function F/. Z is an array with the same shape as R, except possibly along the last axis.

Consecutive sub-arrays along the last axis of Z are defined in terms of the operator Reduce.

If 0<¯1↑⍴R, then Z is:

```
      (1↑[⍴⍴R]R),(F/2↑[⍴⍴R]R), ... ,(F/R)
```

If 0=¯1↑⍴R, then Z is R.

If F is a scalar function, then Z has the same shape as R, and

```
      (I⌷[⍴⍴R] Z)  ↔  F/ I↑[⍴⍴R] R
```

for every simple scalar *I* for which (*I*⌷[⍴⍴*R*]*R*) is defined.

Examples:

```
      +\ 1
1

      +\ 1 2
1 3

      +\ 1 2 3
1 3 6

       +\ 3 4⍴⍳12
1  3  6 10
5 11 18 26
9 19 30 42

      ⍴\ 2 3
2 3 3

      ⍴¨\ 2 3
 2  3 3

      ,\ 'ABCDE'
AABABCABCDABCDE

      ,¨\ 'ABCDE'
 A AB ABC ABCD ABCDE

      ,¨\ 2 3⍴1 2 3 4 5 6
 1  1 2  1 2 3
 4  4 5  4 5 6
```

The symbol ⍀ may be used instead of \ to indicate the first axis of *R* rather than the last.

Example:

```
       +⍀ 3 4⍴⍳12
 1  2  3  4
 6  8 10 12
15 18 21 24
```

Scan with Axis: *Z* ← *F*\[*A*] *R*

F may be any dyadic function which produces a result. *R* must be an array whose sub-arrays along axis *A* are appropriate to the derived function *F*/[*A*]. *A* may be a simple scalar or a one element vector containing an integer axis in *R*. *Z* is an array with the same shape as *R*, except possibly along axis *A*.

Consecutive sub-arrays along axis *A* of *Z* are defined in terms of the operator Reduce.

If 0<(⍴*R*)[*A*], then *Z* is:

```
      (1↑[A]R),[A](F/[A]2↑[A]R),[A] ... ,[A](F/[A]R)
```

If 0=(⍴*R*)[*A*], then *Z* is *R*.

If *F* is a scalar function, then *Z* has the same shape as *R*, and

```
      (I⌷[A] Z)  ←→  F/[A] I↑[A] R
```

for every simple scalar *I* for which (*I*⌷[*A*]*R*) is defined.

This operator is like Scan except that an existing axis in *R* other than the default may be specified.

Examples:

```
      +\[2] 2 3 4⍴⍳24
 1  2  3  4
 6  8 10 12
15 18 21 24

13 14 15 16
30 32 34 36
51 54 57 60
```

The symbol ⍀ may be used instead of \.

BRACKET AXIS OPERATOR

Bracket Axis is a notation for an operator specifying the sub-arrays for the application of a monadic or a dyadic function. It has two forms which are distinguished from each other and from an axis specification by one or two semicolons. An axis specification has no semicolons:

```
Z ←   F[A] R      axis specification producing monadic
Z ← L F[A] R      axis specification producing dyadic

Z ←   F[AZ;AR] R        bracket axis producing monadic
Z ← L F[AZ;AL;AR] R     bracket axis producing dyadic
```

Numeric axis scalars or vectors may be present between the brackets and separated by the semicolons, while the permitted range is determined by the particular function and arguments with which it is used. Any axis vector which is empty is treated as an empty integer axis vector. An axis vector may not contain repetitions. An axis vector in non-increasing order implies a transposition.

The Bracket Axis operator is different from an axis specification. Bracket Axis modifies the behavior of a function in a manner which is consistent, and independent of the specific function. The meaning of axis specification depends on the specific function to which it is applied. There are, however, similarities in some cases:

```
  ⊂[A] R  ↔    ⊂[;A] R
  ⌽[A] R  ↔    ⌽[A;A] R
F¨/[A] R  ↔  F¨/[A;A] R
L /[A] R  ↔  L /[A;;A] R
L +[A] R  ↔  L +[A;;A] R   if (⍴⍴L)<⍴⍴R
L +[A] R  ↔  L +[A;A;] R   if (⍴⍴L)>⍴⍴R
L ⌽[A] R  ↔  L ⌽[A;⍳0;A] R
```

⎕IO is an implicit argument of Bracket Axis.

Bracket Axis (producing Monadic)   Z ← F[AZ;AR] R

F may be any monadic function which produces a result. R must be an array whose sub-arrays specified by AR are appropriate to the monadic function F. Z is an array whose sub-arrays are formed by applying F to sub-arrays of R. Each application of F must produce an identically shaped array, which becomes a sub-array of Z along axes AZ.

If present, AR determines the axes of the sub-arrays of R to which F will be applied. If AR is elided, then this defaults to all axes of R. If present, AZ determines the axes of the sub-arrays of Z into which the axes of the result sub-arrays will be put. If elided, then this defaults to the last axes of Z. This results in the following identity:

```
      F[;] R  ↔  F R
```

If AR is present but empty (meaning scalars of R), then the derived function F[;⍳0] is similar to F¨, except that the sub-arrays resulting from applications of function F must be identically shaped, and they are assembled into an array without any additional depth. Provided that each application of F produces an identically shaped array, then:

```
      F[AZ;AR] R  ↔  ⊃[AZ] F¨ ⊂[AR] R
```

If the collection of selected sub-arrays is empty, then the argument presented to F is ∊⊃⊂[AR]R. If in addition F is defined, then F must contain the system label ⎕FL (see "System Labels" on page 227). If the collection of selected sub-arrays is empty, then F may not be derived, except when directly produced by a defined operator containing the system label ⎕FL.

If the result of an application of F is simple, then Z is simple.

Example:

```
      Z ← ,[;⍳0] 1 2 3
      Z
1
2
3
      ⍴Z
3 1
```

If AR specifies all coordinates of R, then the derived function F[;AR] is the same as F.

Example:

```
      Z ← ,[;1] 1 2 3
      Z
1 2 3
      ⍴Z
3
```

The resulting sub-arrays are put along the last axis of Z, unless specified otherwise by AZ. If given, this item must have the same number of elements as the rank of the result of an application of function F.

Example:

```
      Z ← ,[1;1] 1 2 3
      Z
 1 2 3
      ρZ
1 3
```

The shape of the sub-arrays selected from the argument must be appropriate for the function *F*.

Examples:

```
      R ← 3 4ρ1 2 3 14 6 7 8 5 9 12 11 10
      R
1  2  3 14
6  7  8  5
9 12 11 10

      Z ← ⍋[;1] R
      Z
1 2 3
1 2 3
1 2 3
2 3 1
      ρZ
4 3

      Z ← ⍋[;2] R
      Z
1 2 3 4
4 1 2 3
1 4 3 2
      ρZ
3 4

      Z ← ,[1;1 2] 2 3 4ρι24
      Z
 1  2  3  4
 5  6  7  8
 9 10 11 12
13 14 15 16
17 18 19 20
21 22 23 24
      ρZ
6 4

      Z ← ,[2;1 2] 2 3 4ρι24
      Z
1 5  9 13 17 21
2 6 10 14 18 22
3 7 11 15 19 23
4 8 12 16 20 24
      ρZ
4 6
```

```
      Z ← ,[2;2 1] 2 3 4ρι24
      Z
1 13 5 17  9 21
2 14 6 18 10 22
3 15 7 19 11 23
4 16 8 20 12 24
      ρZ
4 6

      R ← ⍕ 3 4ρι12
      ρR
3 10
      R
1  2  3  4
5  6  7  8
9 10 11 12

      Z ← ⍎[;2] R
      ρZ
3 4
      Z
1  2  3  4
5  6  7  8
9 10 11 12

      Z ← ⊃[;ι0] 'HIM' 'HER'
      Z
HIM
HER
      ρZ
2 3

      Z ← ⊃[1;ι0] 'HIM' 'HER'
      Z
HH
IE
MR
      ρZ
3 2
```

If the collection of sub-arrays is empty, then the argument presented to *F* is ∊⊃⊂[*AR*]*R*:

```
      Z ← ⍕[;2] 0 3ρ0
      ρZ
0 5
```

Bracket Axis (producing Dyadic)   Z ← L F[AZ;AL;AR] R

F may be any dyadic function which produces a result. R must be an array whose sub-arrays specified by AR are appropriate to the dyadic function F. L must be an array whose sub-arrays specified by AL are appropriate to F. Z is an array whose sub-arrays are formed by applying F to (possibly different) sub-arrays of L and R. Each application of F must produce an identically shaped array, which becomes a sub-array of Z along axes AZ.

The collections of sub-arrays of L and R must conform with each other in rank and length, unless a sub-array is the entire array, in which case it will be replicated like a scalar as necessary.

If present, AR determines the axes of the sub-arrays of R to which F will be applied. If AR is elided, then this defaults to all axes of R. If present, AL determines the axes of the sub-arrays of L to which F will be applied. If AL is elided, then this defaults to all axes of L. If present, AZ determines the axes of the sub-arrays of Z into which the axes of the result sub-arrays will be put. If elided, then this defaults to the last axes of Z. This results in the following identity:

```
      L F[;;] R  ↔  L F R
```

If AL and AR are present but empty (meaning scalars of L and R), then the derived function F[;⍳0;⍳0] is similar to F¨, except that the sub-arrays resulting from applications of function F must be identically shaped, and they are assembled into an array without any additional depth. Provided that each application of F produces an identically shaped array, then:

```
      L F[AZ;AL;AR] R  ↔  ⊃[AZ] (⊂[AL] L) F¨ ⊂[AR] R
```

If the collections of selected sub-arrays are empty, then the arguments presented to F are ∊⊃⊂[AL]L and ∊⊃⊂[AR]R. If in addition F is defined, then F must contain the system label ⎕FL (see "System Labels" on page 227). If the collections of selected sub-arrays are empty, then F may not be derived, except when directly produced by a defined operator containing the system label ⎕FL.

If the result of an application of F is simple, then Z is simple.

Example:

```
      Z ← 10 20 30 ,[;⍳0;⍳0] 1 2 3
      Z
10 1
20 2
30 3
      ⍴Z
3 2
```

If *AL* and *AR* specify all coordinates of *L* and *R*, then the derived function *F*[;*AL*;*AR*] is the same as *F*.

Example:

```
      Z ← 10 20 30 ,[;1;1] 1 2 3
      Z
10 20 30 1 2 3
      ⍴Z
6
```

If a sub-array is the entire array, then it will be replicated like a scalar as necessary.

Examples:

```
      Z ← 10 20 30 ,[;⍳0;1] 1 2 3
      Z
10 1 2 3
20 1 2 3
30 1 2 3
      ⍴Z
3 4

      Z ← 10 20 30 ,[;1;⍳0] 1 2 3
      Z
10 20 30 1
10 20 30 2
10 20 30 3
      ⍴Z
3 4

      R ← 3 2⍴'ABCDEF'
      R
AB
CD
EF

      Z ← 7 ⍴[;;2] R
      Z
ABABABA
CDCDCDC
EFEFEFE
      ⍴Z
3 7
```

The resulting sub-arrays are put along the last axis of *Z*, unless specified otherwise by *AZ*. If given, *AZ* must have the same number of elements as the rank of the result of an application of function *F*.

Examples:

```
      Z ← 10 20 30 ,[1;⍳0;1] 1 2 3
      Z
10 20 30
 1  1  1
 2  2  2
 3  3  3
      ⍴Z
4 3

      Z ← 10 20 30 ,[1;1;⍳0] 1 2 3
      Z
10 10 10
20 20 20
30 30 30
 1  2  3
      ⍴Z
4 3
```

The shapes of the sub-arrays selected from the left and right arguments need only conform with respect to the function *F*.

Examples:

```
      L ← 3 4⍴'THEYWANTRAIN'
      L
THEY
WANT
RAIN

      Z ← L ⍳[;1;1] 'AT'
      Z
4 1
2 4
4 4
4 2
      ⍴Z
4 2

      Z ← L ⍳[1;1;1] 'AT'
      Z
4 2 4 4
1 4 4 2
      ⍴Z
2 4
```

```
      Z ← L ⍳[;2;1] 'AT'
      Z
5 1
2 4
2 5
      ⍴Z
3 2

      Z ← L ⍳[1;2;1] 'AT'
      Z
5 2 2
1 4 5
      ⍴Z
2 3

      R ← 2 4⍴'|\-*⌽⍉⊖⍟'
      R
|\-*
⌽⍉⊖⍟
      Z ← 1 2 2 1 ⌷[;⍳0;1] R
      Z
|⍉⊖*
      ⍴Z
4
```

If the collections of sub-arrays are empty, then the arguments presented to *F* are ∊⊃⊂[*AL*]*L* and ∊⊃⊂[*AR*]*R*:

```
      Z ← 4 ↑[;;2] 0 3⍴0
      ⍴Z
0 3
```

PRIMITIVE DYADIC OPERATORS

The primitive dyadic operators take a left operand (which must be either a function or ∘) and a right operand (which may be either a function or an array), and produce a dyadic derived function.

ARRAY PRODUCTS

The array product operator (.) may produce either of two derived functions, depending on the left operand:

outer product	*∘.G*
inner product	*F.G*

where *G* is a dyadic function, and *F* is a dyadic function.

Outer Product: *Z ← L ∘.G R*

The left operand of the operator is the symbol ∘. *G* may be any dyadic function. *L* and *R* may be any arrays whose elements are appropriate to the function *G*.

Sub-arrays of *Z* are created by applying the function *G* between each element in *L* and each element in *R*, in all combinations. Each application of function *G* must produce identically shaped results.

Any axes produced by the function *G* are placed last in *Z*, so that:

```
      ρZ  ↔  (ρL),(ρR),S
```

where *S* is the shape of the result of an application of function *G*. If *G* is a scalar (or pervasive) function, then *S* is empty, and ρ*Z* is (ρ*L*),ρ*R*.

If either argument is empty, then *G* is executed <u>once</u> with arguments ⊂∊⊃*L* and ⊂∊⊃*R*. If in addition *G* is defined, then *G* must contain the system label ⎕*FL* (see "System Labels" on page 227). If either argument is empty, then *G* may not be derived, except when directly produced by a defined operator containing the system label ⎕*FL*.

Examples:

```
      Z ← 10 20 ∘.+ 1 2 3
      ⍴Z
2 3
      Z
11 12 13
21 22 23

      R ← 3 4⍴'THEYWANTRAIN'
      R
THEY
WANT
RAIN
      Z ← 'AT' ∘.= R
      Z
0 0 0 0
0 1 0 0
0 1 0 0

1 0 0 0
0 0 0 1
0 0 0 0
      ⍴Z
2 3 4

      Z ← 10 20 ∘.+ 0⍴0
      ⍴Z
2 0

      Z ← 10 20 ∘.× 13 0 7 ⍴0
      ⍴Z
2 13 0 7

      Z ← 10 20 ∘., 1 2 3
      ⍴Z
2 3 2
      Z
10 1
10 2
10 3

20 1
20 2
20 3

      3 4 ∘.(↑¨) '□□□□' '△△△△△' 'oooooo'
 □□□  △△△  ooo
 □□□□ △△△△ oooo
```

If either argument is empty, then *G* is executed once with arguments ⊂∊⊃*L* and ⊂∊⊃*R*:

```
      Z ← 2 3 4 ∘.ρ 0 1 5ρ0
      ρZ
3 0 1 5 0
```

Inner Product: *Z ← L F.G R*

F may be any dyadic function. *G* may be any dyadic function which produces a result. *L* and *R* may be any arrays whose rows and columns (respectively) are appropriate to the function *G*.

The function *G* is applied between each row (along the last axis) of *L*, and each column (along the first axis) of *R*, in all combinations. Then the derived function *F*/ is applied to each of these results.

Identity:

```
      ρZ  ↔  (¯1↓ρL),(1↓ρR),S
```

where *S* is the shape of the result of an application of functions *F*/ and *G*. The derived function +.× is equivalent to matrix multiplication.

If 0∊¯1↓ρ*L* or 0∊1↓ρ*R*, and also *G* is defined, then *G* must contain the system label ⎕*FL*. If 0∊¯1↓ρ*L* or 0∊1↓ρ*R*, then *G* may not be derived, except when directly produced by a defined operator containing the system label ⎕*FL*. (See "System Labels" on page 227.)

If the result of an application of function *G* is empty along its last axis, and also *F* is defined, then *F* must contain the system label ⎕*ID*. If the result of an application of function *G* is empty along its last axis, then *F* may not be derived, except when directly produced by a defined operator containing the system label ⎕*ID*. (See "System Labels" on page 227.)

Examples:

```
      10 20 30 +.× 1 2 3
140

      (2 3ρ⍳6) +.× 3 4ρ⍳12
38  44  50  56
83  98 113 128
```

```
      (2 3ρι6) ⌊.⌈ 3 4ρι12
1 2 3 4
4 4 4 4

      L ← 4 3ρ'ME YOUME TOO'
      L
ME
YOU
ME
TOO

      L ∧.= 'ME '
1 0 1 0

      L ∨.= 'AGO'
0 0 0 1

      R ← 3 8ρ'SATURDAY7/04/81 JULY 4   '
      R
SATURDAY
7/04/81
JULY 4

      R +.∈ '0123456789'
0 5 1

      (2 3ρ1 0 1 0 1 1) ,./ 3 4ρ'MUCHMORETIME'
MT
UI
CM
HE

MT
OI
RM
EE
```

The APL2 system functions manage objects in the active workspace, the APL2 environment, or resources of the APL2 system. They all have distinguished names which begin with a quad (⎕).

Symbol	Monadic	Pg	Dyadic	Pg
⎕*AT*			Attributes	194
⎕*CR*	Canonical Rep.	182		
⎕*DL*	Delay	182		
⎕*EA*			Execute Alternate	197
⎕*ES*	Event Simulation	183	Event Simulation	196
⎕*EX*	Expunge	185		
⎕*FX*	Fix	187	Fix	198
⎕*NC*	Name Class	188		
⎕*NL*	Name List	190	Name List	199
⎕*SVC*	Sh. Var. Control	190	Sh. Var. Control	200
⎕*SVO*	Sh. Var. Offer	191	Sh. Var. Offer	200
⎕*SVQ*	Sh. Var. Query	191		
⎕*SVR*	Sh. Var. Retract	191		
⎕*SVS*	Sh. Var. State	192		
⎕*TF*	Transfer Form	192	Transfer Form	201

Figure 11. System Functions

Canonical Representation: Z ← ⎕CR R

R may be a simple character scalar or vector which represents the name of a displayable defined function or operator. Z is the simple character matrix of the named object's canonical representation.

The first row of the matrix is the function/operator header. It contains the model statement, followed perhaps by a semicolon and a list of local names separated by blanks.

The remaining rows of the matrix are the lines of the function or operator. They contain no unnecessary blanks except for trailing blanks, and blanks in comments (including the ones immediately preceding the ⍝). A canonical form may contain entirely blank lines. An entirely blank row may represent an empty expression in the function. The last column of a canonical form will not be entirely blank.

Example:

```
     ∇ Z←F R
[1]    Z←1+R×2
     ∇

       Z ← ⎕CR 'F'
       Z
Z←F R
Z←1+R×2
       ⍴Z
2 7
```

Delay: Z ← ⎕DL R

R may be a scalar non-negative real number. A pause of approximately R seconds is invoked. Z is a real scalar containing the number of seconds actually delayed. The pause may be interrupted by a strong interrupt.

Example:

```
       ⎕DL 2
2.00128
```

Event Simulation (Monadic): ⎕*ES R*

R must be a simple character scalar or vector, or a zero or two element integer vector. Event Simulation never returns an explicit result.

If *R* is an empty vector, then no action is taken.

If *R* is a non-empty character vector, then it is displayed, and then an error condition is generated in the expression which invoked the function within which the ⎕*ES* occurs. Normal error handling is initiated except no error message is displayed (except for *R*). The Event Type system variable ⎕*ET* is set to 0 1. If ⎕*ES* is executed from within a defined function or operator, then the event action is generated as though the function or operator were locked or primitive.

Example:

```
     ∇ Z←F R
[1]    ⎕ES (0=R)/'WRONG'
[2]    Z←÷R
     ∇

       F 0
WRONG
       F 0
       ∧

       ⎕ET
0 1
```

If *R* is a two element integer vector, then it is assigned to the Event Type system variable ⎕*ET*, and then an event simulation is generated in the expression which invoked the function. If, in addition, *R* is a legal error event code, then an error message in the current national language is reported (refer to the system variable ⎕*NLT*). Legal error event codes are listed on page 208 in the discussion of the system variable ⎕*ET*.

Examples:

```
     ∇ Z←F R
[1]    ⎕ES (0=R)/5 4
[2]    Z←÷R
     ∇

       F 0
DOMAIN ERROR
       F 0
       ∧

       ⎕ET
5 4

       ⎕NLT ← 'DEUTSCH'

       F 0
UNGUELTIGES ARGUMENT
       F 0
       ∧

       ⎕ET
5 4
```

If *R* is 0 0, then the active workspace is cleared if there is no error trapping associated with the expression.

Example:

```
     ∇ F
[1]    ⎕ES 0 0
     ∇

       F
CLEAR WS

       ⎕ET
0 0

       )FNS
```

If the expression ⎕*ES* 0 0 is trapped with the Execute Alternate (⎕*EA*) system function, then a trapped error is generated with no associated message, ⎕*ET* is set to 0 0, and the workspace is not cleared.

Example:

```
     ∇ F
[1]    ⎕ES 0 0
     ∇

       '⎕ET' ⎕EA 'F'
0 0

       '⎕EM' ⎕EA 'F'

       F
       ∧

       )FNS
F
```

Expunge: Z ← ⎕EX R

R must be a simple character scalar, vector, or matrix whose rows are interpreted as APL2 names. Z is a simple logical scalar or vector of shape ¯1↓⍴R.

Each name in R is disassociated from any value it may have had, <u>if</u> it represented either of:

1. a defined operator

2. a defined function

3. a (non-system) variable, which may or may not be localized in a defined function or operator

4. a system variable which is localized in a defined function or operator, and which is one of ⎕CT, ⎕FC, ⎕IO, ⎕LX, ⎕MD, ⎕PP, ⎕PR, or ⎕RL.

5. an argument of a defined function or operator

6. an operand of a defined operator

If any of the objects are shared variables, then their shares are retracted. Z has shape ¯1↓⍴R, and contains a 1 for each corresponding variable name in R if the name is now available for use.

If a name in R is that of a system label or a system function, then the corresponding element of Z is 0, but the meaning of the name will remain unchanged.

If a name in *R* is that of a system variable, then the corresponding element of *Z* is 1. If in addition the system variable is one of ⎕*CT*, ⎕*FC*, ⎕*IO*, ⎕*LX*, ⎕*MD*, ⎕*PP*, ⎕*PR*, or ⎕*RL*, then its value is removed.

Example:

```
      A ← 1
      ⎕EX 2 1⍴'AB'
1 1
      A
VALUE ERROR
      A
      ∧
```

Example:

```
    ∇ Z←L (F P G) R;X V
[1]  B: V←1
[2]   ⎕NC 9 1⍴'ZLFPGRXVB'
[3]   ⎕EX 9 1⍴'ZLFPGRXVB'
[4]   ⎕NC 9 1⍴'ZLFPGRXVB'
    ∇

      1 +P× 2
0 2 3 4 3 2 0 2 1
1 1 1 1 1 1 1 1 0
0 0 0 0 0 0 0 0 1
```

Suspended or pendent defined functions may be Expunged. This will not, however, change the definition of a previously invoked function in the state indicator. Such a function will retain its original definition until its execution is completed. Until such time, the previously invoked definition exists on the stack only, and it may not be edited.

Example:

```
      ⍎⎕FX 'F' '1' '2÷0' '3'
1
DOMAIN ERROR
F[2] 2÷0
     ∧∧

      )SI
F[2]

      ⎕EX 'F'
1
      )SI
F[2]
```

```
      →3
3
      F
VALUE ERROR
      F
      ∧
```

Fix (Monadic): *Z* ← ⎕*FX R*

R must be a simple character matrix, or a vector of character scalars, vectors, or both. *Z* is either an integer scalar, or a character vector.

R represents the definition for a function or operator in a pseudo canonical form. If the definition is a valid one, then the function or operator is established in the active workspace, and *Z* is the name of the established object. The name of the function or operator that is being established must be either undefined, or the name of another defined function or operator. It may not be the name of a variable.

If the definition is not a valid one, then *Z* is a scalar integer indicating the first row of the function or operator line which is in error. The integer is dependent upon ⎕*IO*.

⎕*FX* will accept a pseudo canonical form with the following variations from a canonical form:

1. It may contain unnecessary blanks.

2. The header may have semicolons between local names instead of blanks.

3. It may be a vector of character scalars and/or vectors instead of a character matrix.

A canonical form may contain entirely blank lines. Trailing blanks in comments will be removed.

Examples:

```
      ⎕FX 'Z←F R' 'Z←1+R×2'
F
      F 3
7

      ⎕FX (⊂'Z←F R'),⊂'Z←1+R×2',⎕AV[1]
2
```

Suspended or pendent defined functions may be Fixed. This will not, however, change the definition of a previously invoked function in the state indicator. Such a function will retain its original definition until its execution is completed. Until such time, the previously invoked definition and the current definition may be different.

Example:

```
      ⍎⎕FX 'F' '1' '2÷0' '3'
1
DOMAIN ERROR
F[2] 2÷0
     ∧∧

     )SI
F[2]

      ⎕FX 'F' '11' '12' '13'
F

      →3
3
      F
11
12
13
```

Name Class: Z ← ⎕NC R

R must be a simple character scalar, vector, or matrix whose rows are interpreted as APL2 names. Z is a simple integer scalar or vector of shape ¯1↓⍴R.

Each element in Z is the name class of the corresponding name in R. A name class may be:

¯1 - invalid name

0 - unused but valid name

1 - name of a label (a constant)

2 - name of a variable

3 - name of a function

4 - name of an operator

An undefined name is classified as a variable if it has been shared (but not yet assigned).

Example:

```
      A ← 1
      0 ⎕SVO 'B'

      ⎕NC 3 1⍴'ABQ'
2 2 0
```

Symbols which represent primitive functions and operators are classified as invalid names.

Example:

```
      ⎕NC 3 1⍴'+/¨'
¯1 ¯1 ¯1
```

The names of system labels, system variables, and system functions are treated like those of common labels, variables, and functions.

Example:

```
      ⎕NC 3 3⍴'⎕ID⎕IO⎕NC'
1 2 3
```

The name class of the name of an argument in a defined function or operator may be 0 or 2, depending upon whether or not it has been used.

Example:

```
     ∇ Z←L F R
[1]    Z←⎕NC 2 1⍴'LR'
     ∇

      F 1
0 2
```

The name class of the name of an operand in a defined operator may be 2 or 3, depending upon whether it is an array or a function.

Examples:

```
    ∇ Z←(F O G) R;V X
[1]  V←1
[2]  Z←⎕NC 6 1ρ'FOGRVX'
    ∇

      +OP× 1
3 4 3 2 2 0

      (+OP 0) 1
3 4 2 2 2 0
```

Name List (Monadic):   Z ← ⎕NL R

R must be a simple integer scalar or vector containing only 1, 2, 3, or 4. Z is a simple character matrix.

Z is a simple character matrix of the names of all objects currently active (and most local) in the workspace whose name class is mentioned in R (see the system function ⎕NC). Names of distinguished system objects (⎕-names) are not included. The list is in alphabetical order, according to the Atomic Vector (⎕AV) character sequence in Figure 17 on page 285.

Example:

```
      A ← AC ← AB ← 1

      ⎕NL 2
A
AB
AC
```

Shared Variable Control (Monadic):   Z ← ⎕SVC R

R must be a simple character scalar, vector, or matrix whose rows are interpreted as APL2 names. Z is a simple logical vector or matrix of shape (¯1↓ρR),4. Z contains the 4-element vector of access controls for each corresponding (non-system) variable name in R. If a name in R is something other than a non-system variable, than the corresponding access control vector is 0 0 0 0.

The access control vector is described with the dyadic Shared Variable Control system function on page 200.

Shared Variable Offer (Monadic): *Z* ← ⎕*SVO R*

R must be a simple character scalar, vector, or matrix whose rows are interpreted as APL2 names. *Z* is an integer scalar or vector of shape ¯1↓ρ*R*. *Z* contains the degree of coupling for each corresponding variable name in *R*.

There are three possible degrees of coupling:

0 - unshared

1 - share offer extended, but not consummated

2 - shared

Shared Variable Query: *Z* ← ⎕*SVQ R*

R may be an empty vector, or a scalar or one element simple integer array. *Z* is an integer vector or a character matrix.

If *R* is an empty vector, then *Z* is an integer vector of identifications of processors making share offers.

If *R* is a scalar or one element integer array containing a processor identification, then *Z* is a matrix of variable names offered by the processor specified by *R*, but not yet shared (degree of coupling less than 2).

Shared Variable Retraction: *Z* ← ⎕*SVR R*

R must be a simple character scalar, vector, or matrix whose rows are interpreted as APL2 names. *Z* is an integer scalar or vector of shape ¯1↓ρ*R*. The degree of coupling for each variable named in *R* is reduced to 0. *Z* contains the previous degree of coupling for each corresponding variable name in *R*. After a shared variable has been retracted, it is not shared.

Shared Variable State: Z ← ⎕SVS R

R must be a simple character scalar, vector, or matrix whose rows are interpreted as APL2 names. Z is a simple logical vector or matrix of shape (¯1↓ρR),4. Z contains the 4-element vector of access states for each corresponding variable name in R. A vector of access states may have any of four possible values:

0 0 0 0 not a shared variable

0 0 1 1 set by one processor, and referenced by the other (also, the initial state)

1 0 1 0 set by first processor, but not yet referenced by the second

0 1 0 1 set by second processor, but not yet referenced by the first

The "second processor" is the one with which the sharing is done (the one mentioned in dyadic ⎕SVO).

Transfer Form (Monadic): Z ← ⎕TF R

R must be a simple character scalar or vector. Z is a simple character vector.

If R is the name of a variable, or a displayable defined function or operator, then Z is a character vector which is the extended transfer form of that object. If the transfer form cannot be formed, then Z is an empty character vector ('').

If R is the name of a shared variable, then taking its transfer form constitutes a reference of the variable.

If R is the extended transfer form of a variable, a defined function, or a defined operator, then that object is established in the workspace, and Z is a character vector containing its name. If the transfer form is invalid, then Z is an empty character vector (''). This is called the Inverse Transfer Form.

Identity:

```
      ⎕TF R  ←→  2 ⎕TF R
```

for all valid R.

⎕*TF* is described in more detail in the discussion of the dyadic system function Transfer Form, and in "Appendix C. The Extended Transfer Form" on page 319.

Attributes: *Z* ← *L* ⎕*AT R*

R must be a simple character scalar, vector, or matrix whose rows are interpreted as APL2 names. *L* must be an integer scalar. *Z* is an integer vector, or matrix. ⁻1↓⍴*Z* is ⁻1↓⍴*R*. *Z* contains an attribute vector for each corresponding object name in *R* according to the integer *L*:

1 - Valences (length 3)

1. explicit result
2. function valence
3. operator valence

2 - Fix Time (length 7)

1. year
2. month
3. day
4. hour
5. minute
6. second
7. millisecond

3 - Execution Properties (length 4)

1. non-displayable
2. non-suspendable
3. ignores weak interrupts
4. converts non-resource errors to *DOMAIN ERROR*

Examples:

```
      ⎕FX 'L FN R' 'L÷R'
FN
      ⎕FX 'Z ←(F OPR) R' 'F ÷R'
OPR
      0 1 1 0 ⎕FX 'L GN R' 'L÷R'
GN
      VAR ← 1 2

      1 ⎕AT 3 3ρ'FN OPRVAR'
0 2 0
1 1 1
1 0 0

      2 ⎕AT 3 3ρ'FN OPRVAR'
1980 7 4 10 13 50 990
1980 7 4 10 13 51  15
   0 0 0  0  0  0   0

      3 ⎕AT 3 3ρ'FN GN OPR'
0 0 0 0
0 1 1 0
0 0 0 0
```

Valences are discussed further in "Function and Operator Definition" on page 275. The fix time for a variable is always all zeros. The execution properties for a variable are always all zeros.

Identity:

```
    ρL ⎕AT R   ↔   (¯1↓ρR),N
```

where *N* is the length of the particular attribute vector specified by *L*.

The results of ⎕AT applied to the names of system objects are:

R is name of ...	1 ⎕AT *R*	2 ⎕AT *R*	3 ⎕AT *R*
System Label	1 0 0	7ρ0	0 0 0 0
(undefined)	0 0 0	7ρ0	0 0 0 0
System Variable	1 0 0	7ρ0	0 0 0 0
(undefined)	0 0 0	7ρ0	0 0 0 0
System Function	1 2 0	7ρ0	1 1 1 0

Event Simulation (Dyadic): *L* ⎕*ES R*

R must be a zero or two element simple integer vector. *L* must be a simple character scalar or vector. Event Simulation never returns an explicit result.

If *R* is an empty vector, then no action is taken.

If *R* is a simple two element vector, then it is assigned to the Event Type system variable ⎕*ET*, *L* is displayed, and then an event simulation is generated in the expression which invoked the function within which the ⎕*ES* occurs. Normal error handling is initiated except no error message is displayed (except for *L*). If ⎕*ES* is executed from within a defined function or operator, then the event action is generated as though the function or operator were locked or primitive.

Dyadic Event Simulation is like monadic Event Simulation except that both the message to be reported and the new value of the Event Type system variable ⎕*ET* may be specified.

Example:

```
    ∇ Z←F R
[1]   'WRONG' ⎕ES (0=R)/13 17
[2]   Z←÷R
    ∇

      F 2
0.5

      F 0
WRONG
      F 0
      ∧

      ⎕ET
13 17
```

If *R* is 0 0, then *L* is reported and the active workspace is cleared if there is no error trapping associated with the expression.

Example:

```
      ∇ F
[1]     'NOT AUTHORIZED' ⎕ES 0 0
      ∇

        F
NOT AUTHORIZED
CLEAR WS

        ⎕ET
0 0

        )FNS
```

If the expression *L* ⎕*ES* 0 0 is trapped with the Execute Alternate (⎕*EA*) system function, then a trapped error is generated with message *L*, ⎕*ET* is set to 0 0, and the workspace is not cleared.

Example:

```
      ∇ F
[1]     'NOT AUTHORIZED' ⎕ES 0 0
      ∇

        '⎕ET' ⎕EA 'F'
0 0

        '⎕EM' ⎕EA 'F'
NOT AUTHORIZED
        F
        ∧

        )FNS
F
```

Execute Alternate: *Z* ← *L* ⎕*EA* *R*

R must be a simple character vector or scalar. *L* must be a simple character vector or scalar. Both *L* and *R* must contain only valid APL2 characters, and not any terminal control characters (see "The APL2 Character Set" on page 285).

R is taken to represent an APL2 expression, and is executed in the context of the statement in which it is found. *Z* is the value of the APL2 expression in *R*. If the expression has no value, then *L* ⎕*EA* *R* has no value.

If there is an error in the APL2 expression *R*, or if *R* is interrupted, then execution of *R* is aborted without an error message, and *L* is executed instead. In that case, *Z* is the value of the APL2 expression in *L*. If the expression has no value, then *L* ⎕*EA* *R* has no value. Execution of *L* is subject to normal error handling.

Examples:

```
      'ι2' ⎕EA 'ι4'
1 2 3 4

      'ι2' ⎕EA 'ι4.5'
1 2

      '→' ⎕EA 'ι4.5'

      'ι2.3' ⎕EA 'ι4.5'
DOMAIN ERROR
      ι2.3
      ^
      'ι2.3' ⎕EA 'ι4.5'
      ^      ^
```

If *R* calls a defined function *F*, then the statements executed by *F* are also under control of the error trap. In particular, *R* could call a long running function, and *L* could be an error recovery function.

Fix (Dyadic): *Z* ← *L* ⎕*FX* *R*

R must be a simple character matrix, or a vector of character scalars, vectors, or both. *L* must be a simple logical 4 element vector. *Z* is either a simple integer scalar, or a simple character vector.

R represents the definition for a function or operator. If the definition is a valid one, then the function or operator is established in the active workspace, and *Z* is the name of the established object. The name of the function or operator that is being established must be either undefined, or the name of another defined function or operator. It may not be the name of a variable.

If the definition is not a valid one, then *Z* is a scalar integer indicating the row of the function or operator line which is in error. The integer is dependent upon ⎕*IO*.

Dyadic Fix is like monadic Fix, except that the defined function or operator is given execution properties as specified by *L*. There are four independent properties:

1. is not displayable

2. is not suspendable

3. is not interruptible

4. converts any non-resource error to *DOMAIN ERROR*

Each property may be set independently by the corresponding element of *L*. Having all four execution properties the same as being locked. See "Execution Properties" on page 277 for a further description and examples of these properties.

A defined function or operator which can be displayed may have its execution properties re-defined by the expression *L ⎕FX ⎕CR R*, where *R* is the name of the function, and *L* is the vector of properties.

Name List (Dyadic): *Z ← L ⎕NL R*

R must be a simple integer scalar or vector containing only 1, 2, 3, or 4. *L* must be a simple character scalar or vector. *Z* is a character matrix.

Z is a matrix of the names of all objects currently active in the workspace whose name class is mentioned in *R*, and the first character of whose name occurs in *L*. Names of distinguished system objects (⎕ names) are not included. The list is in alphabetical order, according to the Atomic Vector (⎕*AV*) character sequence in Figure 17 on page 285.

Example:

```
      A ← AC ← AB ← BB ← 1

      'A' ⎕NL 2
A
AB
AC
```

Refer to the Name Class system variable (⎕*NC*).

Shared Variable Control (Dyadic): Z ← L ⎕SVC R

R must be a simple character scalar, vector, or matrix whose rows are interpreted as APL2 names. L must be a simple logical scalar, vector, or matrix. Z is a simple logical vector or matrix.

If L is a scalar, a 1-element vector, or a 4-element vector, then it is reshaped to shape ((¯1↓ρR),4) before application of the function.

The 4-element access control vectors in L are imposed on the corresponding variables named in R. Z has shape (¯1↓ρR),4, and contains the resulting access controls.

The resulting access controls may be more restrictive than those which were set, because a processor may only increase the degree of control imposed by the other.

Ones in the 4 logical elements of an access control vector are interpreted as follows:

1. Two successive sets by the first processor require an intervening set or reference by the second processor.

2. Two successive sets by the second processor require an intervening set or reference by the first processor.

3. Two successive references by the first processor require an intervening set by the second processor.

4. Two successive references by the second processor require an intervening set by the first processor.

The "second processor" is the one with which the sharing is done (the one mentioned in dyadic ⎕SVO). The access control vector is symmetric in the sense that the controls L of the first processor appear to be controls L[2 1 4 3] to the second processor.

Shared Variable Offer (Dyadic): Z ← L ⎕SVO R

R must be a simple character scalar, vector, or matrix whose rows are interpreted as APL2 names. L must be a simple integer scalar or vector, interpreted as processor identifications. Z is an integer scalar or vector of shape ¯1↓ρR.

If L is scalar, then it is reshaped to shape (¯1↓ρR) before application of the function. If L is non-scalar, then it must have shape (¯1↓⍴R). The variables named in R are offered to the corresponding processors in L. Z contains the resulting degree of coupling for each corresponding variable name in R. Degrees of coupling are described in the discussion of the monadic Shared Variable Offer system function.

If a row of R contains a pair of names, then the first is the name of the variable to be shared, and the second is a surrogate name which is offered to match a name offered by another processor. The name of a variable may be its own surrogate. This is the default. Thus, the following two statements have the same effect:

```
      100 ⎕SVO 'CTL'
      100 ⎕SVO 'CTL CTL'
```

A share offer to processor 0 is taken to mean a general share offer to any processor.

Transfer Form (Dyadic):   Z ← L ⎕TF R

R must be a simple character scalar or vector. L must be a simple integer scalar or one element vector. Z is a simple character vector.

Dyadic Transfer Form is like monadic Transfer Form, except that the type may be specified by the left argument L. There are two types of transfer form:

1. The migration transfer form, not permitted for non-simple variables or defined operators (described in detail in "Appendix B. The Migration Transfer Form" on page 317).

2. The extended transfer form, permitted for any variables and displayable defined functions and operators (described in detail in "Appendix C. The Extended Transfer Form" on page 319).

If R is the name of a variable, or a displayable defined function or operator, then Z is a character vector which is the transfer form of type L for that object. If the transfer form of type L cannot be formed, then Z is an empty character vector ('').

If R is the name of a shared variable, then taking its transfer form constitutes a reference of the variable.

If R is the transfer form (of type L) of a variable, defined function, or defined operator, then that object is established in the workspace, and Z is a character vector containing its name. If the transfer form is invalid, then *Z* is an empty character vector (''). This is called the Inverse Transfer Form.

Inverse Transfer Form ignores name class conflicts. That is, if there is a variable named *X* in the active workspace, an inverse transfer form may be performed to establish a function or operator with the same name *X*. Similarly, if there is a function or operator named *X* in the active workspace, an inverse transfer form may be performed to establish a variable with the same name *X*. Additionally, if there is a shared variable named *X* in the active workspace, and if an inverse transfer form is performed to establish a variable with the same name *X*, then the old variable is expunged before the new variable is formed, so that any share on that variable is retracted.

Identity:

```
      ⎕TF R  ←→  2 ⎕TF R
```

for all valid *R*.

The migration transfer form is not permitted for defined operators, but the inverse migration transfer form is.

The APL2 system variables help manage objects in the active workspace, the APL2 environment, or resources of the APL2 system. They all have distinguished names which begin with a quad (⎕) or a quote quad (⍞).

Global value persists over a)*CLEAR* or a)*LOAD*	Can not be effectively localized	Ignores an assignment	Set by the system upon an error	Symb	Name	Pg
.	.	.	.	⎕*CT*	Comparison Tolerance	206
.	.	.	.	⎕*FC*	Format Control Characters	210
.	.	.	.	⎕*IO*	Index Origin	212
.	.	.	.	⎕*LX*	Latent Expression	216
.	.	.	.	⎕*MD*	Matrix Divide Tolerance	217
.	.	.	.	⎕*PP*	Printing Precision	218
.	.	.	.	⎕*PR*	Prompt Replacement	218
.	.	.	.	⎕*RL*	Random Link	221
.	●	.	.	⍞	Character Input / Output	204
.	●	.	.	⎕	Input / Output	213
.	●	.	.	⎕*SVE*	Shared Variable Event	222
.	●	.	●	⎕*L*	Left Argument	214
.	●	.	●	⎕*R*	Right Argument	220
.	●	●	●	⎕*EM*	Event Message	207
.	●	●	●	⎕*ET*	Event Type	208
.	●	●	.	⎕*AI*	Account Information	204
.	●	●	.	⎕*AV*	Atomic Vector	204
.	●	●	.	⎕*IR*	Implicit Result	212
.	●	●	.	⎕*LC*	Function Line Counter	215
.	●	●	.	⎕*TC*	Terminal Control Characters	223
.	●	●	.	⎕*TS*	Time Stamp	223
.	●	●	.	⎕*TT*	Terminal Type	224
.	●	●	.	⎕*UL*	User Load	224
.	●	●	.	⎕*WA*	Workspace Available	225
●	.	.	.	⎕*HT*	Horizontal Tabs	211
●	.	.	.	⎕*NLT*	National Language Translation	217
●	.	.	.	⎕*PW*	Printing Width	219
●	.	.	.	⎕*TZ*	Time Zone	224

Figure 12. System Variables

Figure 12 shows the system variables grouped by their properties. The system variables for which implicit errors are possible are ⎕*CT*, ⎕*FC*, ⎕*IO*, ⎕*MD*, ⎕*PP*, ⎕*PR*, and ⎕*RL*. System variables whose values are assigned by the system upon an error are called debug variables. They are ⎕*EM*, ⎕*ET*, ⎕*L*, and ⎕*R*.

System variables which do not ignore an assigned value, and whose global value persists over a workspace)CLEAR or)LOAD, are called <u>session</u> <u>variables</u>. They are ⎕HT, ⎕NLT, ⎕PW, and ⎕TZ.

Account Information: ⎕AI

This is a four element simple integer vector reporting:

⎕AI[1] - user identification

⎕AI[2] - compute time (ms)

⎕AI[3] - connect time (ms)

⎕AI[4] - keying time (ms)

Elements of ⎕AI beyond 4 are not defined but are reserved.

The system re-specifies ⎕AI, so that specifying it or localizing it has no effect.

Atomic Vector: ⎕AV

This is a simple character vector of all 256 characters in the APL2 character set (See "The APL2 Character Set" on page 285). The results of displaying or printing certain elements of ⎕AV may depend on the type of terminal or printer being used.

The system re-specifies ⎕AV, so that specifying it or localizing it has no effect.

Character Input/Output: ⍞

This is a variable shared with the system. It may be a simple character array. The behavior depends upon whether it is being assigned or referenced:

When ⍞ is assigned with an array, then the array is displayed at the terminal without the normal ending new line character. Successive assignments of vectors to ⍞ without any other intervening terminal output or input cause attempts to display the arrays on the same terminal line. The sum of the widths of the assignments should be less than the width of the terminal being used.

Example:

```
     ∇ F X
[1]    ⍞←'X '
[2]    ⍞←'='
[3]    ⍞←' '
[4]    ⍞←X
     ∇

      F 13
X = 13
```

When ⍞ is referenced, terminal input is requested, and is returned as a character vector.

A reference to ⍞ which is preceded by an assignment to ⍞ without any other intervening terminal output or input involves a prompt/response interaction. The last (or only) row of the assignment is called the prompt, and the result of the reference is called the response. The response is a vector composite of:

1. a transformation of the unchanged characters in the prompt

2. the terminal input, including changed characters in the prompt

The transformation of unchanged characters in the prompt is determined by the Prompt Replacement system variable (⎕PR).

On display terminals, the prompt is displayed in the entry area of the terminal. It may be appended or changed before input. If ⎕PR is '', then the result of the reference to ⍞ is the vector appearing in the display area when the entry is made.

The sum of the widths of the prompt and the response should be less than the width of the terminal being used, or the result may be unpredictable.

Examples:

```
     ∇ X←F
[1]    ⍞←'X = '
[2]    X←⍞
     ∇
```

```
      ⎕PR ← '*'

      Z ← F
X = 13                {13 is typed by the user}

      ρZ
6
      Z
****13

      ⎕PR ← ''

      Z ← F
X = 13                {13 is typed by the user}

      ρZ
6
      Z
X = 13
```

If a strong attention is signalled while terminal input is requested through ⍞, then an interrupt is generated. On certain terminals, such an interrupt can be generated by overstriking the three letters *O U T*.

Comparison Tolerance: ⎕*CT*

This is a simple real numeric scalar. It is the quantity used to determine fuzzy equality. The value in a clear workspace is 1*E*¯13.

Real numbers *L* and *R* are considered equal if

```
      (|L-R) ≤ ⎕CT×(|L)⌈|R
```

where the above ≤ is strict, and uses no tolerance.

Complex numbers *L* and *R* are considered equal if both their real and imaginary parts are equal. A complex number is considered to be real for comparison purposes if the greater of the absolute values of the imaginary part and the tangent of the angle is much less than ⎕*CT*. Refer to the primitive dyadic function Equal for examples.

The range of ⎕*CT* is 0≤⎕*CT* and ⎕*CT*<1. The implementation of the equality determination is approximate. It is done in such a way that large values of ⎕*CT* (near 1) become meaningless.

⎕*CT* is an implicit argument of the monadic functions Ceiling (⌈), Floor (⌊), and Unique (∩), and the dyadic functions Encode (⊤), Equal (=), Find (⍷), Find Index (⍳), Greater (>), Index of

(⍳), Less (<), Member (∊), Not Equal (≠), Not Greater (≤), Not Less (≥), and Residue (|).

Event Message: ⎕EM

This is a simple character matrix containing the text of the last error or event message. The value in a clear workspace is 3 0⍴' '.

⎕EM contains at least 3 rows:

1. The error message (in the current national language)

2. The statement invoking the error

3. The carets pointing to the statement:

   a. The rightmost caret indicates where the error occurred.

   b. The leftmost caret indicates how far evaluation of the statement had proceeded prior to the error.

4. Possible further information

In some cases, the left and the right caret will both indicate the same position, and only one caret will be seen.

Errors occurring within the functions Execute or Execute Alternate result in messages which contain more than 3 rows.

If there is not enough room in the workspace (⎕WA) to form ⎕EM at the time of the error, but there is room to suspend the statement, then ⎕EM will be a character matrix of shape 3 0, and the error event type code ⎕ET will not be affected.

The system re-specifies ⎕EM, so that specifying it or localizing it has no effect. ⎕EM is automatically local to a function called by a line entered in immediate execution. If there is not enough room in the workspace (⎕WA) to suspend the statement in error, then *WS FULL* will be reported, ⎕EM will be set to a character matrix of shape 3 0,

Example:

```
      ⍳4.5
DOMAIN ERROR
      ⍳4.5
      ∧
      ⍴⎕EM
3 12
      ⎕EM
DOMAIN ERROR
      ⍳4.5
      ^
```

Example:

```
      ⍎ '⍳4.5'
DOMAIN ERROR
      ⍳4.5
      ^
      ⍎'⍳4.5'
      ^

      ⍴⎕EM
5 13

      ⎕EM
DOMAIN ERROR
      ⍳4.5
      ^
      ⍎'⍳4.5'
      ^
```

Event Type: ⎕*ET*

This is a simple two element integer vector. It is set by the system to the event type code of the most recent event. The value in a clear workspace is 0 0.

The first element of ⎕*ET* gives the major classification of the event type code, and the second element gives a sub-class:

0 N - Defaults

    0 0 - no error
    0 1 - unclassified event (⎕*ES* '??')

1 N - Resource Errors

1 1	-	*INTERRUPT*
1 2	-	*SYSTEM ERROR*
1 3	-	*WS FULL*
1 4	-	*SYSTEM LIMIT* of symbol table
1 5	-	*SYSTEM LIMIT* of no shares
1 6	-	*SYSTEM LIMIT* of interface quota
1 7	-	*SYSTEM LIMIT* of interface capacity
1 8	-	*SYSTEM LIMIT* of array rank
1 9	-	*SYSTEM LIMIT* of array size
1 10	-	*SYSTEM LIMIT* of array depth
1 11	-	*SYSTEM LIMIT* of prompt length

2 N - *SYNTAX ERROR*

2 1	-	no array (2×)
2 2	-	ill-formed line ([(])
2 3	-	name class (3←2)
2 4	-	illegal operation in context ((*A*←*B*)←2)

3 N - *VALUE ERROR*

3 1	-	name with no value
3 2	-	function with no result

4 N - Implicit Argument Errors

4 1	-	⎕*PP ERROR*
4 2	-	⎕*IO ERROR*
4 3	-	⎕*CT ERROR*
4 4	-	⎕*FC ERROR*
4 5	-	⎕*RL ERROR*
4 6	-	⎕*MD ERROR*
4 7	-	⎕*PR ERROR*

5 N - Explicit Argument Errors

5 1	-	*VALENCE ERROR*
5 2	-	*RANK ERROR*
5 3	-	*LENGTH ERROR*
5 4	-	*DOMAIN ERROR*
5 5	-	*INDEX ERROR*
5 6	-	*AXIS ERROR*

All undefined major event classifications numbered 0 through 99 are reserved. Refer to "Error Messages" on page 253 for more information about particular errors.

The system re-specifies ⎕*ET*, so that specifying it or localizing it has no effect. ⎕*ET* is automatically local to a function called by a line entered in immediate execution. If there is not enough room in the workspace (⎕*WA*) to suspend the statement in error, then *WS FULL* will be reported, ⎕*EM* will be set to a character matrix of shape 3 0, and ⎕*L* and ⎕*R* will not be set.

⎕*ET* may be set because of execution of the system function Event Simulation ⎕*ES*.

Format Control: ⎕*FC*

This is a simple 6-element character vector containing control characters implicitly used by the functions monadic and dyadic Format, and default array display. The value in a clear workspace is '.,*0_*J*'.

The element definitions are:

⎕*FC*[1] - use for decimal point

⎕*FC*[2] - use for comma

⎕*FC*[3] - fill when otherwise blank for digit 8

⎕*FC*[4] - fill when otherwise *DOMAIN ERROR* for overflow

⎕*FC*[5] - print as blank (may not be ,.0123456789)

⎕*FC*[6] - complex number formatting (*J*, *R*, or *D*)

Elements of ⎕*FC* beyond 6 are not defined but are reserved.

⎕*FC*[1] is used wherever a decimal point is needed in Picture Format:

```
      ⎕FC[1] ← ','
      '5.5555' ⍕ 3.1415
3,1415
```

⎕*FC*[2] is used wherever a comma is needed in Picture Format:

```
      ⎕FC[2] ← '.'
      '555,555,555' ⍕ 123456789
123.456.789
```

⎕*FC*[3] is used to fill where a field containing an 8 would otherwise be blank in Picture Format:

```
      ⎕FC[3] ← '⎕'
      '855555' ⍕ 1234
⎕⎕1234
```

⎕*FC*[4] is used to fill where a field is too small for a number or a non-scalar item in dyadic Format:

```
      ⎕FC[4] ← '?'
      '5555' ⍕ 123456
????
      4 0 ⍕ 123456
????
      4 0 ⍕ 1234 'SEVEN'
1234????
```

If ⎕*FC*[4] is '0' (which is the default), then a field which is too small will result in a *DOMAIN ERROR*.

⎕*FC*[5] is replaced by a blank without ending a field wherever it is used in Picture Format:

```
      ⎕FC[5] ← '⊖'
      '$⊖355' ⍕ 12
$ 12
```

⎕*FC*[6] specifies either *J*, *R*, or *D* formatting for complex numbers in default numeric displays or monadic Format:

```
      ⎕FC[6] ← 'J'
      ⍕ 0J1
0J1

      ⎕FC[6] ← 'R'
      ⍕ 0J1
1R1.570796327

      ⎕FC[6] ← 'D'
      ⍕ 0J1
1D90
```

Horizontal Tabs: ⎕*HT*

This is a simple non-negative integer scalar or vector containing horizontal tab settings for typewriter terminals. (The left margin position is counted as position 1.)

On input, tab characters are translated to an appropriate number of consecutive blanks. On output, consecutive blanks may be translated to tab characters. The value of ⎕*HT* must accurately reflect the physical tab settings of the terminal being used, or terminal input may not be accepted, and misleading or unreadable output may be produced.

⎕*HT* is a session variable. That is, if a valid global value is assigned, then it will persist over a workspace clear or load. If an invalid value is assigned, then it is ignored by the system. The initial value at the beginning of a session is ⍳0.

Index Origin: ⎕*IO*

This is a simple integer scalar containing the index of the first element of any non-empty vector. The value in a clear workspace is 1. The only acceptable values are 0 or 1.

⎕*IO* is an implicit argument of the monadic functions Fix (⎕*FX*), Grade Down (⍒), Grade Up (⍋), Index Set (⌷), Interval (⍳), Roll (?), the dyadic functions Deal (?), Find Index (⍸), Fix (⎕*FX*), Grade Down (⍒), Grade Up (⍋), Index (⌷), Index of (⍳), Pick (⊃), Transpose (⍉), Bracket Indexing, and bracketed axes.

Implicit Result: ⎕*IR*

This is an array containing the implicit result of a function. The value in a clear workspace is 0. ⎕*IR* is set by the system when certain primitive functions execute.

For the functions Matrix Inverse (⌹*R*) or Matrix Divide (*L*⌹*R*), if ⎕*MD* is 0, then the implicit result is the number of independent rows in the matrix (the algebraic rank). ⎕*MD* is set whether or not the function executes without a *DOMAIN ERROR*. Refer to the descriptions of these functions for more information.

The system re-specifies ⎕*IR*, so that specifying it or localizing it has no effect.

Input / Output: ⎕

This behaves like a variable shared with the system. The behavior depends upon whether it is being assigned or referenced:

When ⎕ is assigned an array, the array is displayed at the terminal.

Example:

```
      ⍴⎕←⍳3
1 2 3
3
```

When ⎕ is referenced, a prompt (⎕:) is displayed at the terminal, and terminal input is requested under control of default error or interrupt handling. After the requested input is supplied and evaluated (by producing an array), error and interrupt handling reverts to whatever it was prior to the reference of ⎕.

Examples:

```
      ⍴A←10+⎕×2
⎕:
      ⍳3
3
      A
12 14 16

      ⍴A←10+⎕×2
⎕:
      ⍳3.4
DOMAIN ERROR
      ⍳3.4
      ∧∧
⎕:
      ⍳3
3
      A
12 14 16
```

If the response to a ⎕: prompt is an abort statement (→), then execution will be aborted.

Examples:

```
      ⍴A←10+⎕×2
⎕:
      →
```

```
    ∇ F
[1]   ⍴A←10+⎕×2
    ∇

      F
⎕:
      →

      )SI
```

System commands may be entered when a ⎕: prompt is displayed. Any response to a system command is not treated as a response to ⎕.

Examples:

```
      10+⎕×2
⎕:
      )WSID
IS CLEAR WS
⎕:
      ⍳3
12 14 16

      10+⎕×2
⎕:
      )CLEAR
CLEAR WS
```

Left Argument: ⎕L

This is a variable shared with the system. It is the array value of the left argument of a function whose execution was interrupted by an error. ⎕L has no value in a clear workspace.

⎕L is set whenever an error occurs in a primitive dyadic function. It is effectively automatically local to a function called by a line entered in immediate execution, and exists only while the statement in error is suspended. ⎕L has no value after an error in a monadic function.

Example:

```
     ∇ Z←F R
[1]    Z←(R×1 2)+3 4 5
     ∇

       F 10
LENGTH ERROR
F[1] Z←(R×1 2)+3 4 5
       ∧     ∧

       ⎕L
10 20

       ⎕L ← 10 20 30
       →⍳0
13 24 35
```

If there is not enough room in the workspace (⎕WA) to suspend the statement in error, then *WS FULL* will be reported, ⎕EM will be set to a character matrix of shape 3 0, and ⎕L and ⎕R will not be set.

Refer to the description of system variable ⎕R for further information about error handling.

Line Counter: ⎕LC

This is a simple integer vector of line numbers of defined functions and operators in execution or halted (suspended or pendent), with the most recently activated line number first. The value in a clear workspace is ⍳0. The system re-specifies ⎕LC, so that specifying it or localizing it has no effect.

Examples:

```
      ∇ G
[1]     1
[2]     2
[3]     H
      ∇

      ∇ H
[1]     10
[2]     ⎕LC
[3]     30
      ∇

        G
1
2
10
2 3
30
```

If a function is halted, then there is one element of ⎕LC for each line of the display reported by)SI or)SINL which contains a name (each line which is not immediate execution).

Example:

```
      ∇ Z←F R
[1]     Z←(R×1 2)+3 4 5
      ∇

      F 10
LENGTH ERROR
F[1] Z←(R×1 2)+3 4 5
       ∧      ∧

      ⎕LC
1

      )SI
F[1]
*
```

Latent Expression: ⎕LX

This may be any simple character scalar or vector representing an *APL* statement that is automatically executed (by ⍎⎕LX) whenever the workspace is activated (with the)*LOAD* system command). It may be used to display a message, to invoke an arbitrary function, or to resume an interrupted program. The value in a clear workspace is ''.

Matrix Divide Tolerance: ⎕*MD*

This may be any simple scalar non-negative real number. It is the tolerance implicitly used in determining the algebraic rank of a matrix in the functions Matrix Divide (*L*⌹*R*) and Matrix Inverse (⌹*R*). Refer to the descriptions of these functions for more details. The value in a clear workspace is 0.

If ⎕*MD* is set to a non-zero value, and *R* is a singular matrix, then a pseudo inverse will be computed by ⌹*R*.

National Language Translation: ⎕*NLT*

This may be a simple character vector representing the name of a national language. It determines in what language error messages will be reported. It also indicates what language in addition to the default will be accepted for system commands (See "Appendix I. National Language Translations" on page 335). The initial value at the beginning of a session is installation dependent.

The meaningful values are:

'*DANSK*'	Danish
'*DEUTSCH*'	German
'*ENGLISH*'	English
'*ESPANOL*'	Spanish
'*FRANCAIS*'	French
'*NORSK*'	Norwegian
'*SUOMI*'	Finnish
'*SVENSKA*'	Swedish

If ⎕*NLT* is set to other than one of these character vectors, then English is assumed.

''	English
'*MARTIAN*'	English

Leading and trailing blanks are ignored in a setting of ⎕*NLT*. Refer to the discussion of the monadic Event Simulation system function on page 183 for an example.

⎕*NLT* is a session variable. That is, if a valid global value is assigned, then it will persist over a workspace clear or load.

If an invalid value is assigned, then it is ignored by the system, and ⎕*NLT* is reset to '', which defaults to English.

Printing Precision: ⎕*PP*

This is a simple positive integer scalar. It is the number of significant digits in the default display of numbers. The value in a clear workspace is 10.

The minimum value for ⎕*PP* is 1. The maximum value for ⎕*PP* is 18. If ⎕*PP* is 18, then all available precision will be displayed.

⎕*PP* is an implicit argument of the function monadic Format (⍕).

Prompt Replacement: ⎕*PR*

This is a simple character scalar, or vector of shape 0 or 1. It controls the interaction between an assignment (the prompt), and a successive reference (the response) of the Character Input/Output system variable (⍞). The interaction is such that the first part of the response vector is a transformation of the prompt. The second part is the terminal input. The prompt, as assigned, either may or may not be returned as part of the response, depending on the value of ⎕*PR*. The value in a clear workspace is 1⍴' ', which means to replace unchanged elements of the prompt with blanks.

If ⎕*PR* is a character scalar or 1-element vector, then the prompt is not returned as part of the response. Instead, unchanged elements of the prompt are replaced by the character in ⎕*PR*.

If ⎕*PR* is an empty vector (''), then unchanged elements of the prompt are returned as part of the response.

Refer to the Character Input/Output system variable on page 204 for examples.

Printing Width: □PW

This is a simple positive integer scalar. It controls the presentation of system output. Its minimum value is 30.

If an attempt is made to display an array wider than □PW, then the display will be folded at or just before the □PW width. The folded parts will each be indented six spaces. The folded parts will each be separated from the first part by N blank lines, where N is 0⌈¯1+ρρA.

Examples:

```
      □PW ← 30

      30ρ'Δ'
ΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔ

      31ρ'Δ'
ΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔ
      Δ

      32ρ'Δ'
ΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔ
      ΔΔ
```

Rows of a matrix are folded together:

```
      ⍉32 2ρ'○□'
○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○
□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

      ○○
      □□
```

Planes of a multi-dimensional array are folded together:

```
      ⍉ 32 2 2ρ'○□∘Δ'
○○○○○○○○○○○○○○○○○○○○○○○○○○○○○○
∘∘∘∘∘∘∘∘∘∘∘∘∘∘∘∘∘∘∘∘∘∘∘∘∘∘∘∘∘∘

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
ΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔ


      ○○
      ∘∘

      □□
      ΔΔ
```

The display of a simple array containing numbers may be folded at a width less than ⎕PW so that individual numbers are not split.

Example:

```
      2 3 ∘.⍟ 10 20 30
3.321928095 4.321928095
2.095903274 2.726833028

      4.906890596
      3.095903274
```

If ⎕PW is small and ⎕PP is large, then the display of some complex numbers in a simple array may be split. If ⎕PW is at least 13+2×⎕PP, then individual numbers in a simple array will not be split. Numbers in a non-simple array may be split with any value of ⎕PP.

The value of ⎕PW has no effect on the display of system messages, or on the result of the primitive monadic Format function (⍕).

⎕PW is a session variable. That is, if a valid global value is assigned, then it will persist over a workspace clear or load. If an invalid value is assigned, then it is ignored by the system. The initial value at the beginning of a session is dependent on the type of terminal being used.

Right Argument: ⎕R

This is a variable shared with the system. It is the array value of the right argument of a function whose execution was interrupted by an error which was not a *SYNTAX ERROR* or a *VALUE ERROR*. ⎕R has no value in a clear workspace.

⎕R is set whenever an error occurs in a primitive function. It is effectively automatically local to a function called by a line entered in immediate execution, and exists only while the statement in error is suspended.

Example:

```
    ∇ Z←F R
[1]   Z←(R×1 2)+3 4 5
    ∇

      F 10
LENGTH ERROR
F[1] Z←(R×1 2)+3 4 5
       ∧      ∧
      ⎕R
3 4 5
      ⎕R ← 3 4
      →⍳0
13 24
```

Note that the branch expression →⍳0 caused the suspended function + to be restarted at the point of the error with the new value of the right argument.

Everything in the statement to the right of the leftmost caret in the error report was already evaluated prior to the error, and was not evaluated again after the branch →⍳0. This may be especially important if the statement in error contains shared variables or defined functions or operators.

If there is not enough room in the workspace (⎕*WA*) to suspend the statement in error, then *WS FULL* will be reported, ⎕*EM* will be set to a character matrix of shape 3 0, and ⎕*L* and ⎕*R* will not be set.

Random Link: ⎕*RL*

This is a simple positive integer scalar not greater than ¯2+2*31. It is used and set implicitly by the functions Roll and Deal (?). The value in a clear workspace is 16807.

Example:

```
      ⎕RL
16807
      ?2
1
      ⎕RL
282475249
```

Repeatable results can be obtained from the functions Roll and Deal if ⎕*RL* is set to a particular value first.

Example:

```
      ⎕RL ← 13
      ? 10 100 1000 10000
1 71 823 9625
      ? 10 100 1000 10000
10 85 612 8253

      ⎕RL ← 13
      ? 10 100 1000 10000
1 71 823 9625
      ? 10 100 1000 10000
10 85 612 8253
```

Shared Variable Event: ⎕*SVE*

This is a simple non-negative scalar integer shared with the system. The behavior depends upon whether it is being assigned or referenced:

If ⎕*SVE* is assigned a positive value *N*, then a shared variable event will be scheduled to occur after approximately *N* seconds. If ⎕*SVE* is assigned 0, then a shared variable event will not be scheduled to occur.

If ⎕*SVE* is referenced, then execution is suspended until the occurrence of a shared variable event. The suspension may be interrupted by a strong interrupt.

After a shared variable event has occurred, execution is resumed, and a value is returned indicating the number of seconds remaining in the previously specified (positive) wait time. If the previously assigned time was 0, then the returned result is 0.

When ⎕*SVE* releases after being referenced, any shared variable event previously scheduled by a ⎕*SVE* assignment is cancelled.

The action of ⎕*SVE* is such that if specified with a non-zero value, then referencing it invokes a wait with a time out.

Shared variable events include the changing of the state of any shared variable. Localizing ⎕*SVE* has no effect. The value in a clear workspace is 0. The value in a freshly loaded workspace is 0.

Terminal Control Characters: ⎕*TC*

This is a simple three element character vector.

The element definitions are:

⎕*TC*[1] - backspace

⎕*TC*[2] - new line (carriage return)

⎕*TC*[3] - line feed

The system re-specifies ⎕*TC*, so that specifying it or localizing it has no effect. Elements of ⎕*TC* beyond 3 are not defined but are reserved.

Time Stamp: ⎕*TS*

This is a simple 7-element integer vector.

The element definitions are:

⎕*TS*[1] - the current year

⎕*TS*[2] - the current month

⎕*TS*[3] - the current day

⎕*TS*[4] - the current hour

⎕*TS*[5] - the current minute

⎕*TS*[6] - the current second

⎕*TS*[7] - the current millisecond

The value of ⎕*TS* depends on the value of the Time Zone system variable (⎕*TZ*).

Example:

```
      ⎕TS
1981 7 4 19 13 17 210
```

The system re-specifies ⎕*TS*, so that specifying it or localizing it has no effect.

Terminal Type: ⎕*TT*

This is a simple integer scalar. It is a code for the type of terminal in use.

The possible values are:

0 - Indeterminate

1 - Correspondence

2 - PTTC / BCD

4 - 3270 with APL feature

5 - 3270 without APL feature

The system re-specifies ⎕*TT*, so that specifying it or localizing it has no effect.

Time Zone: ⎕*TZ*

This is a simple real numeric scalar. It is the difference in hours between local time and Greenwich mean time (GMT). It may be fractional.

The value of ⎕*TZ* affects the times reported by ⎕*TS* and various system commands. ⎕*TS* must be in the range ¯11<⎕*TZ* and ⎕*TZ*≤13. For example, ¯5 is Eastern Standard Time, and ¯8 is Pacific Standard Time.

⎕*TZ* is a session variable. That is, if a valid global value is assigned, then it will persist over a workspace clear or load. If an invalid value is assigned, then it is ignored by the system. The initial value at the beginning of a session is installation dependent.

User Load: ⎕*UL*

This is a simple non-negative integer scalar. On systems where it can be determined, it is the number of users on the system. Otherwise it is 0.

The system re-specifies ⎕*UL*, so that specifying it or localizing it has no effect.

Workspace Available: ⎕*WA*

This is a simple non-zero integer scalar. It is the available space in the active workspace given as the number of bytes or characters it could hold. Due to the nature of the APL2 implementation, the value of ⎕WA can be different in situations that appear the same.

The system re-specifies ⎕*WA*, so that specifying it or localizing it has no effect.

SYSTEM LABELS

The APL2 system labels identify variants of a defined function or operator. When a variant is called, the function or operator is activated with relevant arguments beginning at the corresponding system label (which may or may not be statement 1). Subsequent execution proceeds normally.

System labels may appear at the beginning of any statement in a defined function or operator. A system label does not affect the execution of the statement on which it appears. If a variant is not called, then a system label has no effect other than that of a normal label. A defined function or operator may have more than one system label if they are different.

System labels have distinguished names which begin with a quad (⎕).

Symbol	Name	Pg
⎕*FL*:	Fill	228
⎕*ID*:	Identity	231

Figure 13. System Labels

Fill: ⎕FL:

This label denotes a fill variant of a defined function or defined operator. It is related to valence, which is discussed further in "Function and Operator Definition" on page 275. In the following discussion:

```
F1 is a defined function with valence 1 0
F2 is a defined function with valence 2 0
O11 is a defined operator with valence 1 1
O12 is a defined operator with valence 1 2
O21 is a defined operator with valence 2 1
O22 is a defined operator with valence 2 2
F is a function operand of an operator
G is a function or array operand of an operator
```

The fill variant is activated in six cases:

1. When a monadic function is applied through the Each operator to an empty array:

   ```
   F1¨ R
   F O11¨ R
   F O12 G¨ R
   ```

   where 0∊⍴R

   When activated at ⎕FL, the argument of the (derived) function is ⊃R.

2. When a dyadic function is applied through the Each operator to empty arrays:

   ```
   L F2¨ R
   L F O21¨ R
   L F O22 G¨ R
   ```

   ```
   where 0∊⍴L and 0∊⍴R
      or 0∊⍴L and 0=⍴⍴R
      or 0∊⍴R and 0=⍴⍴L
   ```

   When activated at ⎕FL, the arguments of the (derived) function are ∊⊃L and ∊⊃R.

3. When a monadic function is applied through the Bracket Axis operator to an empty selection of arrays:

   ```
   F1[AZ;AR] R
   F O11[AZ;AR] R
   (F O12 G)[AZ;AR] R
   ```

   where 0∊(~(⍳⍴⍴R)∊AR)/⍴R

When activated at ⎕*FL*, the argument of the (derived) function is ∊⊃⊂[*AR*]*R*.

4. When a dyadic function is applied through the Bracket Axis operator to an empty selection of arrays:

```
L F2[AZ;AL;AR] R
L F O21[AZ;AL;AR] R
L (F O22 G)[AZ;AL;AR] R
```

   where 0∊(~(⍳⍴⍴*R*)∊*AR*)/⍴*R* or 0∊(~(⍳⍴⍴*L*)∊*AL*)/⍴*L*

   When activated at ⎕*FL*, the arguments of the (derived) function are ∊⊃⊂[*AL*]*L* and ∊⊃⊂[*AR*]*R*.

5. When a dyadic function is applied through the outer product operator to an empty array:

```
L ∘.F2 R
L ∘.(F O21) R
L ∘.(F O22 G) R
```

   where 0∊⍴*L* or 0∊⍴*R*

   When activated at ⎕*FL*, the (derived) function is executed once, with arguments ⊂∊⊃*L* and ⊂∊⊃*R*.

6. When a dyadic function is applied through the inner product operator to certain empty arrays:

```
L F2.F2 R
L F2.(F O21) R
L F2.(F O22 G) R
```

   where 0∊¯1↓⍴*L* or 0∊1↓⍴*R*

   When activated at ⎕*FL*, the right operand (derived) function is executed once, with arguments ∊⊃⊂[(0<⍴⍴*L*)/⍴⍴*L*]*L* and ∊⊃⊂[(0<⍴⍴*R*)/1]*R*.

If *F1*, *F2*, *O11*, *O12*, *O21*, or *O22* above does not contain a fill expression denoted by the system label ⎕*FL*, then its described application through the Each, bracket axis, or outer product operators to an empty array causes a *DOMAIN ERROR*.

Example:

```
    ∇ Z←L RESHAPE R
[1]  Z←L⍴R
[2]  →0
[3] ⎕FL: Z←L⍴R+10
    ∇
```

```
      TΔRESHAPE ← 1 3

      Z ← 5 RESHAPE[;;2] 2 2⍴1 2 3 4
RESHAPE[1] 1 2 1 2 1
RESHAPE[1] 3 4 3 4 3
      ⍴Z
2 5
      Z
1 2 1 2 1
3 4 3 4 3

      Z ← 5 RESHAPE[;;2] 1 2⍴1 2
RESHAPE[1] 1 2 1 2 1
      ⍴Z
1 5
      Z
1 2 1 2 1

      Z ← 5 RESHAPE[;;2] 0 2⍴0
RESHAPE[3]
      ⍴Z
0 0
      Z
```

Trace control (TΔ) is described in "Debug Controls" on page 281.

Example:

```
     ∇ Z←L JOIN R
[1]   Z←L,7 7 7,R
[2]   →0
[3] ⎕FL: Z←L,8 8 8,R
     ∇

      TΔJOIN ← 1 3

      Z ← 1 2 ∘.JOIN 3 4 5
JOIN[1] 1 7 7 7 3
JOIN[1] 1 7 7 7 4
JOIN[1] 1 7 7 7 5
JOIN[1] 2 7 7 7 3
JOIN[1] 2 7 7 7 4
JOIN[1] 2 7 7 7 5
      ⍴Z
2 3 5
      Z
1 7 7 7 3
1 7 7 7 4
1 7 7 7 5

2 7 7 7 3
2 7 7 7 4
2 7 7 7 5
```

```
      Z ← 1 2 ∘.JOIN 0⍴0
JOIN[3] 0 8 8 8 0
      ⍴Z
2 0 5

      Z ← (0⍴0) ∘.JOIN 3 4 5
JOIN[3] 0 8 8 8 0
      ⍴Z
0 3 5

      Z ← (0⍴0) ∘.JOIN 0⍴0
JOIN[3] 0 8 8 8 0
      ⍴Z
0 0 5
```

Identity: ⎕*ID*:

This label denotes an identity variant of a dyadic defined function, or defined operator whose derived function is dyadic. It is activated when a function is applied through the Reduce operator to an empty array:

```
      F2/ R
      F2/[A] R
      F O21/ R
      F O21/[A] R
      (F O22 G)/ R
      (F O22 G)/[A] R
```

where:

0=¯1↑⍴*R* or 0=(⍴*R*)[*A*], as appropriate
F2 is a defined function with valence 2 0
O21 is a defined operator with valence 2 1
O22 is a defined operator with valence 2 2
F is a function operand of an operator
G is a function or array operand of an operator

Valences are discussed further in "Function and Operator Definition" on page 275.

If *F2*, *O21*, or *O22* above does not contain an identity expression denoted by the system label ⎕*ID*, then its described reduction of an empty array causes a *DOMAIN ERROR*. Identity expressions for primitive functions are given in the discussion of the Reduce operator on page 160.

When activated at ⎕*ID*, the right argument of the (derived) function is the empty array *R*, and the left argument is the axis *A* of reduction (explicit or default).

The following example mimics the primitive function Divide, except that the identity element is 7.

```
    ∇ Z←L DIVIDE R
[1]  Z←L÷R
[2]  →0
[3] ⎕ID: Z←((L≠⍳⍴⍴R)/⍴R)⍴7
    ∇

      T∆DIVIDE ← 1 3

      DIVIDE/ 2⍴3
DIVIDE[1] 1
1

      DIVIDE/ 1⍴3
3

      DIVIDE/ 0⍴3
DIVIDE[3] 7
7

      DIVIDE/ 3 2⍴3
DIVIDE[1] 1 1 1
1 1 1

      DIVIDE/ 3 1⍴3
3 3 3

      DIVIDE/ 3 0⍴3
DIVIDE[3] 7 7 7
7 7 7
```

Trace control (T∆) is described in "Debug Controls" on page 281.

Reductions of empty arrays are implied by certain inner products *F.G*, where the result of an application of function *G* is empty along its last axis. Such inner products require that if *F* is a defined function, or is directly derived by a defined operator then it must contain the system label ⎕*ID*.

Example:

```
      (2 0⍴0) DIVIDE.× 0 3⍴0
DIVIDE[3] 7
DIVIDE[3] 7
DIVIDE[3] 7
DIVIDE[3] 7
DIVIDE[3] 7
DIVIDE[3] 7
7 7 7
7 7 7
```

The following example uses the identity element 7 in the result of any reduction of an empty array through the defined operator *IE*. The shape of the result is determined in the same way as for pervasive functions (the rank of *R* is reduced).

```
    ∇ Z←L (F IE) R
[1]  Z←L F R
[2]  →0
[3] ⎕ID: Z←((L≠⍳⍴⍴R)/⍴R)⍴7
    ∇

      T∆IE ← 1 3

      +IE/ 2⍴3
IE[1] 6
6

      +IE/ 1⍴3
3

      +IE/ 0⍴3
IE[3] 7
7

      +IE/ 3 2⍴3
IE[1] 6 6 6
6 6 6

      +IE/ 3 1⍴3
3 3 3

      +IE/ 3 0⍴3
IE[3] 7 7 7
7 7 7
```

System commands control the APL2 session. They are prefixed by a right parenthesis. They may be invoked only from the keyboard and not from within defined functions or operators.

In the descriptions that follow, the system commands are given in English, and brackets ([]) indicate that the enclosed item is optional to the system command. System commands in other national languages will be accepted after an appropriate setting of the National Language Translation system variable (⎕*NLT*). The various possible trouble reports are described in more detail in "Error Messages" on page 253.

	IMPROPER LIBRARY REFERENCE	INCORRECT COMMAND	NOT COPIED	NOT FOUND	NOT ERASED	NOT SAVED, ...	SI WARNING	SYSTEM LIMIT	WS FULL	WS LOCKED	WS NOT FOUND
)CLEAR		●									
)CONTINUE		●									
)COPY	●	●	●	●			●	●	●	●	●
)DROP	●	●						●			●
)EDITOR		●									
)ERASE		●			●						
)FNS		●									
)IN		●	●	●				●	●		
)LIB	●	●						●			
)LOAD	●	●						●	●	●	●
)MSG		●									
)MSGN		●									
)NMS		●									
)OFF		●									
)OPR		●									
)OPRN		●									
)OPS		●									
)OUT		●	●					●	●		
)PBS		●									
)PCOPY	●	●	●	●			●	●	●	●	●
)RESET		●									
)QUOTA		●									
)SAVE		●				●		●	●		
)SI		●									
)SINL		●									
)SIS		●									
)SYMBOLS		●									
)VARS		●									
)WSID		●									

Figure 14. System Commands and Common Trouble Reports

)*CLEAR* [size]

This command activates a clear workspace having no name, and gives the report *CLEAR WS*. The previous active workspace is lost. That is, all shares are retracted, the contents of the active workspace are discarded, and the system variables ⎕*CT*, ⎕*EM*, ⎕*ET*, ⎕*FC*, ⎕*IO*, ⎕*IR*, ⎕*L*, ⎕*LC*, ⎕*LX*, ⎕*MD*, ⎕*PP*, ⎕*R*, ⎕*RL*, and ⎕*SVE* are set to standard initial values. (Unless stated otherwise, all examples given in this manual assume the standard initial values of these system variables.)

On some systems, the size of the clear workspace may be specified.

)*CONTINUE* [*HOLD*]

This command replaces a stored private workspace named *CONTINUE* with a copy of the active workspace, and then performs)*OFF* [*HOLD*]. When the next APL2 session is begun, the workspace named *CONTINUE* is automatically loaded. Some systems permit this startup behavior to be modified by an appropriate option.

)*COPY* [library] workspace [:[password]] [names]

This command brings all or selected global APL2 objects (variables, defined functions, and defined operators) from a stored workspace with the given name [in the given library, and having the given password]. A stored workspace is one which has been previously stored with the system command)*SAVE*.

If the name list is not included in the command, then all objects in the stored workspace are copied. If the active workspace contains objects with the same name as any that are copied, the old ones are replaced. If the old objects are not to be replaced, then use the system command)*PCOPY*.

If any objects are successfully copied, the system reports *SAVED*, followed by the time, date, and time zone when the stored workspace was last saved.

If the name list includes objects that are not found in the stored workspace, then *NOT FOUND*: is reported, followed by a list of such names.

If the name list includes objects that that will not fit in the active workspace, then *WS FULL* and *NOT COPIED*: are reported, followed by a list of such names.

If the name list includes the name of a simple character scalar, vector, or matrix enclosed within parentheses, then its rows are interpreted as APL2 names, and these objects are copied instead of the matrix itself. The matrix may, however, contain its own name. This is called Indirect Copy. It offers a convenient way to copy a group of objects simultaneously. Indirect lists may not be nested.

Example:

```
      ONE←2 3⍴'ONETWO'
      TWO←1
      THIS←2
      THAT←3
      GROUP←2 4⍴'THISTHAT'
      THESE←1 2 3
      MORE←3 5⍴'MORE THESETHOSE'
      )SAVE MINE
  10.13.50   7/04/80 (GMT-5)
      )CLEAR
CLEAR WS
      )COPY MINE ONE (GROUP) (MORE)
SAVED  10.13.50   7/04/80 (GMT-5)
      )VARS
MORE    ONE     THAT    THESE   THIS
```

If the name list includes indirectly listed names which are either invalid, or are not found in the stored workspace, then *NOT FOUND* is not reported for those objects.

If a copied object replaces a shared variable, then its share is retracted. If a copied object replaces a defined function or operator which is pendent or suspended then *SI WARNING* is reported.

Stop and Trace controls are not copied.

If there is no workspace with the given name [in the given library], then *WS NOT FOUND* is reported, and the active workspace is not affected. If there is a workspace with the given name, but the given password is incorrect, then *WS LOCKED* is reported, and the active workspace is not affected. If the workspace is not locked, then password must be omitted in the command, and the colon (:) may be omitted.

A library specification must be a positive integer. The library number defaults to your private library. (See "Appendix H. APL2 Under CP/CMS" on page 333.) If you are ineligible to use the specified library, then *IMPROPER LIBRARY REFERENCE* is reported, and the active workspace is not affected.

)*DROP* [library] workspace [:[password]]

This command removes the copy of the stored workspace with the given name [in the given library] if it exists, and if you are eligible to drop it. You do not need a password to drop a locked workspace. On some systems, you may specify a write password.

If the workspace is successfully dropped, the system reports the current time, date, and time zone.

A library specification must be a positive integer. The library number defaults to your private library. (See "Appendix H. APL2 Under CP/CMS" on page 333.) If there is no workspace with the given name [in the given library], then *WS NOT FOUND* is reported. If you are ineligible to drop workspaces from the specified library, then *IMPROPER LIBRARY REFERENCE* is reported.

)*EDITOR* [editor]

If an editor number is supplied, then this command sets up subsequent invocations by the ∇ or ⍫ characters of the specified editor:

1 - the default APL2 editor

2 - the extended APL2 editor, with full screen display processing

If an editor number is not supplied, then the number of the editor currently in force is reported.

The editor number is a session parameter. That is, it will persist over a workspace clear or load. The initial setting is *EDITOR* 1.

)*ERASE* names

This command erases the global objects (variables, defined functions, or defined operators) that are specified in the name list from the active workspace. If any of the objects are shared variables, then their shares are retracted.

If the name list includes objects that are either invalid or not found, then *NOT ERASED*: is reported, followed by a list of such names.

If the name list includes the name of a simple unshared character matrix enclosed within parentheses, then the rows of the matrix are interpreted as APL2 names, and these objects are erased instead of the matrix itself. The matrix may, however, contain its own name. This is called Indirect Erase. It offers a convenient way to erase a group of objects simultaneously. Indirect lists may not be nested.

Example:

```
      ONE←2 3⍴'ONETWO'
      TWO←2
      THREE←3
      THIS←2
      GROUP←2 4⍴'THISTHAT'
      )ERASE ONE (GROUP)
      )VARS
GROUP   THREE   TWO
```

If the name list includes indirectly listed names which are either invalid, or are not found in the active workspace, then *NOT ERASED* is not reported for those objects.

If the name list includes a defined function or operator which is pendent or suspended then *SI WARNING* is not reported. Suspended or pendent defined functions may be erased. This will not, however, change the definition of a previously invoked function in the state indicator. Such a function will retain its original definition until its execution is completed. Until such time, the previously invoked definition exists on the stack only, and it may not be edited.

Example:

```
      ⍎⎕FX 'F' '1' '2÷0' '3'
1
DOMAIN ERROR
F[2] 2÷0
     ∧∧

     )SI
F[2]

      )ERASE F
1
     )SI
F[2]
```

```
      →3
3
      F
VALUE ERROR
      F
      ∧
```

The command)*ERASE* with no name list does nothing.

)*FNS* [first [last]]

This command lists the names of global defined functions that are in the active workspace in alphabetical order. (Alphabetical order is determined by the Atomic Vector (⎕*AV*) character sequence in Figure 17 on page 285). All names reported in the list begin at a character position which is a multiple of eight, so that a multiple-row list will form columns. The optional arguments first and last specify at what points to begin and end a partial list.

Examples:

```
      )FNS
MONTHLY QUARTERLY       TEST    TRY     WEEKLY  YEARLY

      )FNS TR
TRY     WEEKLY  YEARLY

      )FNS T W
TEST    TRY     WEEKLY
```

If the specified first word alphabetically follows the specified last word, then no list will be reported.

Example:

```
      )FNS W T
```

)*IN* filename [list]

This command reads a transfer file of the given name containing the transfer forms of APL2 objects, and defines those objects locally in the active workspace. The optional list argument specifies what objects to transfer. The default is to transfer all of the objects in the file. The names of successfully transferred objects are not reported.

If the specified transfer file does not exist, or if the file exists but it is not a transfer file, then no objects will be transferred, and *NOT FOUND* will be reported. If any objects are specifically requested but are not found in the transfer file, then *NOT FOUND*: is reported, followed by a list of such names. If any objects are specifically requested but have an invalid transfer form on the file, then *NOT COPIED*: is reported, followed by a list of such names.

The name of a transfer file depends on the operating system (See "Appendix H. APL2 Under CP/CMS" on page 333). The transfer file must contain encodings of transfer forms, like those produced by the)*OUT* system command, or by the *MIGRATE* workspace (See "Appendix D. Migration To/From APL2" on page 323).

```
)LIB [library] [:[password]] [first [last]]
```

This command lists names of workspaces in the specified library. A library specification must be a positive integer. It defaults to your private library. (See "Appendix H. APL2 Under CP/CMS" on page 333.) All names reported in the list begin at a character position which is a multiple of eight, so that a multiple-row list will form columns. The optional arguments first and last specify at what points to begin and end a partial list. On some systems, you may specify a read password.

If you are ineligible to use the specified library, or if it doesn't exist, then *IMPROPER LIBRARY REFERENCE* is reported.

```
)LOAD [library] workspace [:[password]] [size]
```

This command activates a copy of a stored workspace which has the given name [in the given library, and with the given password]. A stored workspace is one which has been previously stored with the system command)*SAVE*.

On some systems, the size of the loaded workspace may be specified. If the workspace size is specified, then there may not be a space between the colon and the password.

If the workspace is successfully activated, the system reports *SAVED*, followed by the time, date, and time zone when the workspace was last saved. Also reported is the size of the workspace being loaded, and the size of the workspace when it was last saved (in parentheses). Both sizes include both the used and the unused space.

Loading a workspace retracts any previous variable shares.

If there is no workspace with the given name [in the given library], then *WS NOT FOUND* is reported, and the active workspace is not affected. If there is a workspace with the given name [in the given library], but the given password is incorrect, then *WS LOCKED* is reported, and the active workspace is not affected. If the workspace is not locked, then the password must be omitted in the command, and the colon (:) may be omitted.

A library specification must be a positive integer. The library number defaults to your private library. (See "Appendix H. APL2 Under CP/CMS" on page 333.) If you are ineligible to use the specified library, then *IMPROPER LIBRARY REFERENCE* is reported, and the active workspace is not affected.

```
    )MSG user message
```

This command attempts to send an arbitrary message to another user. If the attempt was successful, then *SENT* will be reported. If the attempt was unsuccessful, then *NOT SENT* will be reported.

After sending a message, some terminals will prevent you from making further keyboard entries until you receive a reply message from the user, or until a weak interrupt is issued.

Messages can be received any time your terminal is displaying output. They can, however, be suppressed by issuing the system command)*MSG OFF*. If this is done, then any user attempting to send you a message will receive the report *NOT SENT*. The system command)*MSG ON* will restore the acceptance of any messages sent to you. Temporary reception of messages is permitted during the time that keyboard entries are inhibited after a message is sent, even if)*MSG OFF* has been issued.

```
    )MSGN user message
```

This command attempts to send an arbitrary message to another user, but does not wait for a reply (see the)*MSG* command).

```
)NMS [first [last]]
```

This command lists the names of global objects (variables, defined functions, and defined operators) that are in the active workspace in alphabetical order, along with an indication of the name class of each. (Alphabetical order is determined by the Atomic Vector (⎕*AV*) character sequence in Figure 17 on page 285). Each name reported is followed by a dot, and an integer indicating its name class: 2 for variable, 3 for defined function, and 4 for defined operator. These numbers are the same as those produced by the Name Class system variable (⎕*NC*). The optional arguments first and last specify at what points to begin and end a partial list.

Examples:

```
      )NMS
ASSESS.3        COST.2  DAY.2   WEEK.2  WEEKLY.4

      )NMS CAN DO
COST.2  DAY.2
```

If the specified first word alphabetically follows the specified last word, then no list will be reported.

Example:

```
      )NMS DO ASSESS
```

```
)OFF [HOLD]
```

This command terminates the APL2 session and the host session. If *HOLD* is included in the command, then only the APL2 session is terminated, and control is returned to the host. (See "Appendix H. APL2 Under CP/CMS" on page 333.)

```
)OPR message
```

This command attempts to send an arbitrary message to the system operator (see the)*MSG* command).

```
)OPRN message
```

This command attempts to send an arbitrary message to the system operator, but does not wait for a reply (see the *)MSG* command).

```
)OPS [first [last]]
```

This command lists names of global defined operators that are in the active workspace in alphabetical order. (Alphabetical order is determined by the Atomic Vector (⎕*AV*) character sequence in Figure 17 on page 285). The optional arguments first and last specify at what points to begin and end a partial list. (Refer to the examples given with the description of the *)FNS* system command).

```
)OUT filename [list]
```

This command writes the transfer form of objects in the active workspace to a transfer file. The optional list argument specifies what objects to transfer. The default is to transfer all of the unshared variables, defined functions, and defined operators in the workspace. Objects other than these are not transferable. The local meaning of such objects is what is transferred. The names of successfully transferred objects are not reported.

If any objects are specifically requested but are not found in the active workspace, or are not appropriate for transfer, then *NOT COPIED*: is reported, followed by a list of such names.

System variables may be transferred with *)OUT* if specifically requested.

Example:

```
      )OUT SV ⎕CT ⎕IO
```

The name of a transfer file depends on the operating system (See "Appendix H. APL2 Under CP/CMS" on page 333). The transfer file produces by *)OUT* contains encodings of transfer forms suitable for reading by the *)IN* system command, or by the *MIGRATE* workspace (See "Appendix D. Migration To/From APL2" on page 323).

```
)PBS [character]
```

This command reports or specifies the printable backspace character. This character is translated to a backspace in terminal input in certain contexts, so that new members of the APL2 character set may be entered from certain terminals which do not have them or the backspace. On output, the new APL2 characters are translated to their common overstrike combinations, with the backspace showing as the specified character. The printable backspace character is effective only on buffered terminals.

The printable backspace character may not be a blank. If the character is not supplied as part of the command, then the printable backspace character is reported. The command *)PBS OFF* will remove the printable backspace character.

The affected new APL2 characters are:

⌻	⎕*AV*[116]	quad jot
⍸	⎕*AV*[117]	iota underbar
⍷	⎕*AV*[118]	epsilon underbar
⌷	⎕*AV*[205]	squad
⍂	⎕*AV*[207]	quad backslash
≡	⎕*AV*[226]	equal underbar
∵	⎕*AV*[237]	dotted del

Example:

```
      )PBS I
      A ← '⎕I∘ ⍳I_ ∊I_ [I] ⎕I\ =I_ .I¨ '
      ⍴A
14
      A
⎕I∘ ⍳I_ ∊I_ [I] ⎕I\ =I_ .I¨
```

The overstrike pairs may be entered in either order, with an intervening printable backspace.

Example:

```
      B ← '∘I⎕ _I⍳ _I∊ ]I[ \I⎕ _I= ¨I. '
      A ∧.= B
1
```

The printable backspace character is effective only in the context of these new APL2 characters.

Example:

```
      C ← '↑I↓'
      ⍴C
3
      C
↑I↓

      )PBS &
      A
⎕&∘ ⍳&_ ∊&_ [&] ⎕&\ =&_ .&¨
      )PBS OFF
      A
⌻ ⍸ ⍷ ⌷ ⍂ ≡ ∵
      C
↑I↓
```

The printable backspace character is a session parameter. That is, it will persist over a workspace clear or load. The initial setting is *OFF*.

)PCOPY [library] workspace [:[password]] [names]

This command brings all or selected global APL2 objects (variables, defined functions, and defined operators) from a stored workspace with the given name [in the given library, and having the given password]. A stored workspace is one which has been previously stored with the system command *)SAVE*.

This command is the same as *)COPY*, except that if the active workspace contains objects with the same name as any that are requested to be copied, they are not copied, and the old ones are not replaced. The names of such directly specified objects are reported following the message *NOT COPIED*:. The names of such indirectly specified objects are not reported. Refer to the description of the *)COPY* system command for more details.

)QUOTA

This command reports information about the availability of your private library, workspaces, and shared variables, in the form:

```
LIB       TOTAL  FREE REMAINING
WS      DEFAULT  MAX     MAXIMUM
SV       NUMBER  SIZE       SIZE
```

Where:

TOTAL is the total amount of space (in bytes) that is in your library.

REMAINING is the remaining amount of space (in bytes) that is left in your library for saving.

DEFAULT is the default size (in bytes) of the active workspace.

MAXIMUM is the maximum size workspace (in bytes) that may be requested (as with *)CLEAR* or *)LOAD*).

NUMBER is the maximum number of variables which may be simultaneously shared.

SIZE is the size (in bytes) of the shared memory.

Example:

```
      )QUOTA
LIB   2400000  FREE    1800000
WS     380928  MAX      380928
SV         88  SIZE      32768
```

)RESET

This command clears all suspended and pendent statements and editing sessions from the state indicator. This is equivalent to entering Abort statements (→) until the state indicator is clear. *)RESET* also purges and contracts the internal symbol table.

Because of their effective localization to functions in lines of immediate execution, the system variables Event Message (⎕*EM*) and Event Type (⎕*ET*) are reset to their initial values in a clear workspace. Also, the values of the system variables Left Argument (⎕*L*) and Right Argument (⎕*R*) are removed.

)*SAVE* [[library] workspace [:[password]]]

This command stores a copy of the active workspace with the given name [in the given library, and optionally with the given password]. Current values of any shared variables are saved in the stored copy. If the workspace is successfully saved, the system responds with the time, date, and time zone when the workspace was saved, and the workspace name if it was omitted from the command.

If the library number, workspace name, or password are omitted, then they are supplied from the current workspace identification. (See the)*WSID* command, and "Appendix H. APL2 Under CP/CMS" on page 333.)

)*SAVE* also purges and contracts the internal symbol table.

)*SI*

This command reports the current state indicator. This is a list of the calling sequence of defined functions and operators (and their pertinent line numbers) which led to the current state. The report includes one line for each suspended or pendent defined function or operator.

Immediate execution statements are indicated in the list by a star (*). Suspended defined functions or operators are either at the top of the list, or just below a star in the list. Other defined functions and operators in the list are pendent.

Examples:

```
      ∇F
[1]   1
[2]   G
[3]   3
      ∇

      ∇G
[1]   3÷0
      ∇

        F
1
DOMAIN ERROR
G[1] 3÷0
      ∧∧
      )SI
G[1]
F[2]
*

      1 2+3 4 5
LENGTH ERROR
      1 2+3 4 5
      ∧  ∧

      )SI
*
G[1]
F[2]
*
```

)SINL

This command reports the current state indicator, and name lists of those names local to each suspended or pendent defined function or operator. (See also the)SI system command.)

Example:

```
    ∇F;A B
[1]  A←1
[2]  G
[3]  B←1
    ∇

    ∇G;C D E
[1] L:3÷0
    ∇

      F
DOMAIN ERROR
G[1] 3÷0
     ∧∧

      )SINL
G[1] C D E L
F[2] A B
*
```

)*SIS*

This command reports the current state indicator, and also lists the statements which were being executed at the invocation of each line, as well as an indication of where in the statements execution has proceeded. (See also the)*SI* system command.)

Example:

```
   ∇X←F
[1]  1
[2]  X←G×2
[3]  3
   ∇

   ∇Z←G
[1]  Z←3÷0
   ∇

     1+F
1
DOMAIN ERROR
G[1] 3÷0
    ∧∧

     )SIS
G[1]  Z←3÷0
        ∧∧
F[2]  X←G×2
        ∧∧
*  1+F
   ∧∧
```

)*SYMBOLS* [number]

If the number is specified, then this command expands or contracts the internal symbol table to at least the given number of slots. The symbol table is automatically expandable, but system efficiency may be improved by enlarging the it with the)*SYMBOLS* command. A larger symbol table consumes more workspace, but may save computation time. Some workspace may be reclaimed by contracting the symbol table.

If the number is not specified, then this command purges and contracts the internal symbol table, and reports the number of symbols currently in use. This is larger than the number of names of variables, functions, and operators in use.

```
   )VARS [first [last]]
```

This command reports the names of global defined variables that are in the active workspace in alphabetical order. (Alphabetical order is determined by the Atomic Vector (⎕*AV*) character sequence in Figure 17 on page 285). The optional arguments first and last specify at what points to begin and end a partial list. (Refer to the examples given with the description of the)*FNS* system command).

```
   )WSID [library] [workspace] [:[password]]
```

If the workspace [and optional library and password] is specified, then this assigns the given name [library and password] to the active workspace. A workspace name must be composed of alphanumeric characters, and must begin with an alphabetic character. It may be further qualified by the operating system in use. A library specification must be a positive integer. The library number defaults to your private library. (See "Appendix H. APL2 Under CP/CMS" on page 333.)

If no arguments are specified, then this command reports the library (if not your own) and the name of the active workspace (without the password).

Examples:

```
      )WSID
CLEAR WS

      )WSID FIRST
WAS CLEAR WS

      )WSID 2 SECOND :MYKEY
WAS FIRST

      )WSID
2 SECOND
```

ERROR MESSAGES

The APL2 system reports errors with a message.

If the error is the result of an attempted invalid execution of a primitive, then the default error handling is to halt execution, and report:

1. the error message

2. the statement which caused the error

3. An indication of where in the statement the error occurred

At that point, execution may be resumed or aborted after possible corrective action. The system variables Event Message (⎕*EM*) and Event Type (⎕*ET*) will contain further information about the error. Refer to the discussion of the system variable ⎕*R* on page 220 for more details about error recovery.

Error handling other than the default may be requested by:

1. assigning attributes to defined functions or operators with the dyadic system function Fix (⎕*FX*)

2. using the system function Execute Alternate (⎕*EA*)

3. using the system function Event Simulation (⎕*ES*)

The following discussion refers only to the error messages in English. Error messages in other national languages can be produced after an appropriate setting of the system variable National Language Translation (⎕*NLT*). See "Appendix I. National Language Translations" on page 335.

```
AXIS ERROR
CLEAR WS
DEFN ERROR
DOMAIN ERROR
ENTRY ERROR
IMPROPER LIBRARY REFERENCE
INCORRECT COMMAND
INDEX ERROR
INTERRUPT
LENGTH ERROR
LIBRARY I/O ERROR
LIBRARY IN USE, RETRY
NOT AN APL2 WS
NOT COPIED
NOT ERASED
NOT FOUND
NOT SAVED, THIS WS IS CLEAR WS
NOT SAVED, THIS WS IS ________
NOT SAVED, WS QUOTA USED UP
RANK ERROR
SI WARNING
SYNTAX ERROR
SYSTEM ERROR
SYSTEM LIMIT
VALENCE ERROR
VALUE ERROR
WS FULL
WS LOCKED
WS NOT FOUND
⎕__ ERROR
```

Figure 15. Common Error Reports

AXIS ERROR

This report is given upon an attempt to either:

> Execute a primitive function or operator with an axis specification, but the axis specification is incompatible with respect to the particular operation and its argument(s).
>
> Execute the Bracket Axis operator, but an axis specification is incompatible with respect to the particular function and its argument(s).
>
> Execute ⎕*ES* 5 6

Examples:

```
      10+[7] 1 2
AXIS ERROR
      10+[7] 1 2
      ∧ ∧

      ⌽[3] 2 3⍴'ME YOU'
AXIS ERROR
      ⌽[3] 2 3⍴'ME YOU'
      ∧

      10+[;;7] 1 2
AXIS ERROR
      10+[;;7] 1 2
      ∧ ∧

      10+[;] 1 2
AXIS ERROR
      10+[;] 1 2
      ∧ ∧

      10+[;;;;] 1 2
AXIS ERROR
      10+[;;;;] 1 2
      ∧ ∧
```

CLEAR WS

This report is given when either:

> An APL2 session is begun, and the workspace *CONTINUE* is not automatically loaded.

)*CLEAR* is executed.

A *SYSTEM ERROR* occurs.

⎕*ES* 0 0 is executed.

The active workspace is cleared. That is, all shares are retracted, the contents of the active workspace are discarded, and the system variables ⎕*CT*, ⎕*EM*, ⎕*ET*, ⎕*FC*, ⎕*IO*, ⎕*IR*, ⎕*L*, ⎕*LC*, ⎕*LX*, ⎕*MD*, ⎕*PP*, ⎕*PR*, ⎕*R*, ⎕*RL*, and ⎕*SVE* are set to their standard initial values.

DEFN ERROR

This report is given upon an attempt to either:

Enter an invalid ∇ or ⍫ command (to enter an editor).

Enter an invalid edit command (while in an editor).

Leave an editor by establishing an object which is invalid.

Execute ⎕*ES* 6 1

Examples:

```
      ∇ 19
DEFN ERROR
      ∇ 19
      ∧

      F ∇ G
DEFN ERROR
      F ∇ G
      ∧
```

DOMAIN ERROR

This report is given upon an attempt to either:

Execute a primitive function or operator when the requested calculation is beyond the range of the system implementation, and it does not fall into one of the categories of *SYSTEM LIMIT*. This can occur with some of the mathematical functions.

Execute a primitive function with incompatible data type, degree of nesting, or range of the argument(s) with respect to the function.

Execute a primitive operator with incompatible operand(s), or argument(s), or both with respect to the operator.

Execute a defined function or operator which has the error conversion execution property and which invokes any non-resource error (see "Function and Operator Definition" on page 275).

Execute ⎕*ES* 5 4

Examples:

```
      ÷'ME'
DOMAIN ERROR
      ÷'ME'
      ∧∧

      ⌹1 (2 3)
DOMAIN ERROR
      ⌹1 (2 3)
      ∧∧

      ¯6⍴9
DOMAIN ERROR
      ¯6⍴9
      ∧  ∧

      1¨2
DOMAIN ERROR
      1¨2
      ∧∧
```

ENTRY ERROR

This report is given when an invalid character in an input statement has been transmitted to or received by the APL2 system. The system prompts with the received valid characters, and permits the positions of the invalid characters to be filled before re-entry.

IMPROPER LIBRARY REFERENCE

This report is given upon an attempt to issue either of the system commands *)COPY*, *)LIB*, *)LOAD*, *)PCOPY*, or *)SAVE* with a non-existent or ineligible library number.

INCORRECT COMMAND

This report is given upon an attempt to issue any invalid system command, or any valid system command with invalid arguments.

Examples:

```
      )WHY
INCORRECT COMMAND

      )WSID 13
INCORRECT COMMAND
```

INDEX ERROR

This report is given upon an attempt to either:

> Perform Bracket Indexing, Index (⌷), or Pick (⊃), where the index is invalid with respect to the array being indexed.
>
> Execute ⎕*ES* 5 5

Examples:

```
      1 2 3[¯6]
INDEX ERROR
      1 2 3[¯6]
      ∧     ∧

      18⌷1 2 3
INDEX ERROR
      18⌷1 2 3
      ∧ ∧
      3J4⊃1 2 3
INDEX ERROR
      3J4⊃1 2 3
      ∧  ∧
```

INTERRUPT

This report is given when either:

An interrupt or attention has been received by the APL2 system during processing. If so, execution is halted as if an error had occurred.

⎕ES 1 1 is executed.

Example:

```
      +\⍳1E4
      {strong attention from terminal}
INTERRUPT
      +\⍳1E4
      ∧
```

Execution may be resumed with the Branch expression.

LENGTH ERROR

This report is given upon an attempt to either:

Execute a primitive function or operator whose argument(s) have incompatible length(s) with respect to each other or to the operation.

Execute ⎕ES 5 3

Examples:

```
      1 2 3+10 20
LENGTH ERROR
      1 2 3+10 20
      ∧    ∧

      1 2 3⍴¨'AB'
LENGTH ERROR
      1 2 3⍴¨'AB'
      ∧    ∧
```

If the function is dyadic pervasive, and the arguments are nested arrays, then the error may be at a level below the top.

Example:

```
      (1 2)(3 4 5)+(10 20 30)(40 50)
LENGTH ERROR
      (1 2)(3 4 5)+(10 20 30)(40 50)
      ^            ^
```

LIBRARY I/O ERROR

This report is given when one of the library system commands *)CONTINUE*, *)COPY*, *)DROP*, *)LOAD*, *)PCOPY*, or *)SAVE* is used, but an internal error prevents successful completion of the operation.

LIBRARY IN USE, RETRY

This report is given on some systems when one of the library system commands *)CONTINUE*, *)COPY*, *)DROP*, *)LOAD*, *)PCOPY*, or *)SAVE* is used, but another user has (temporary) control of a shared library, and thus prevents successful completion of the operation.

NOT AN APL2 WS

This report is given upon an attempt to use the system command *)LOAD* to load something which is other than an APL2 workspace. For example a workspace dumped to file because of a *SYSTEM ERROR* may be copied from, but not loaded.

NOT COPIED

This report is given when either:

> An object in a name list specified with the system command *)PCOPY* already exists in the active workspace.

An object in a name list specified with the system command)*COPY* or)*PCOPY* does not fit in the active workspace.

An object in a name list specified with the system command)*IN* has an invalid transfer form in the transfer file.

An object in a name list specified with the system command)*OUT* is not transferable.

The report is followed by a colon and a list of such names.

NOT ERASED

This report is given when an object in a name list specified with the system command)*ERASE* cannot be found in the active workspace. The report is followed by a colon and a list of such names.

NOT FOUND

This report is given when either:

An object in a name list specified with one of the system commands)*COPY* or)*PCOPY* cannot be found in the specified stored workspace.

An object in a name list specified with the system command)*IN* cannot be found in the specified transfer file.

The file specified with the system command)*IN* cannot be found, or is not a transfer file.

In the first two cases, the report is followed by a colon and a list of such names.

NOT SAVED, THIS WS IS CLEAR WS

This report is given upon an attempt to issue the system command)*SAVE* when the active workspace has no name.

Example:

```
      )CLEAR
CLEAR WS
      )SAVE
NOT SAVED, THIS WS IS CLEAR WS
```

NOT SAVED, THIS WS IS ________

This report is given upon an attempt to issue the system command *)SAVE* name when the given name conflicts with the name of a stored workspace, and the name is not the name of the active workspace. The report is followed by the name of the active workspace.

Example:

```
      )WSID THATONE
WAS CLEAR WS
      )SAVE
  19.50.10   7/04/81 (GMT-4) THATONE
      )WSID THISONE
WAS THATONE
      )SAVE THATONE
NOT SAVED, THIS WS IS THISONE
```

NOT SAVED, LIBRARY FULL

This report is given upon an attempt to issue the system command *)SAVE* when the allotted storage space for saved workspaces has already been filled.

RANK ERROR

This report is given upon an attempt to either:

> Execute a primitive function or operator whose argument(s) have incompatible rank(s) with respect to each other or to the operation.
>
> Execute *⎕ES* 5 2

Examples:

```
      10 20+3 4⍴⍳12
RANK ERROR
      10 20+3 4⍴⍳12
      ^    ^

      '*o',¨2 3⍴'ME YOU'
RANK ERROR
      '*o',¨2 3⍴'ME YOU'
      ^   ^
```

If the function is dyadic pervasive, and the arguments are nested arrays, then the error may be at a level below the top.

Example:

```
      (1 2)(3 4 5)+(2 3⍴10 20 30)(40 50 60)
RANK ERROR
      (1 2)(3 4 5)+(2 3⍴10 20 30)(40 50 60)
      ^           ^
```

SI WARNING

This report is given when either:

> A suspended or pendent defined function or operator is replaced or severely altered by editing, or by either of the system commands *)COPY* or *)PCOPY*.
>
> An attempt is made to resume (with →⍳0) a function which is not resumable (see below).

In the case of alteration through editing, *)COPY*, or *)PCOPY*, there are two possible consequences:

1. The function is restartable, but not resumable. That is, it may be continued at the beginning of a line from immediate execution by entering either →⎕*LC*, or →*L* (where *L* is a line number), but not in the middle of a line by entering →⍳0. Elements of ⎕*LC* corresponding to such damaged functions or operators are not changed. Lines reported by *)SI*, *)SINL*, and *)SIS* corresponding to such damaged functions or operators show negative line numbers.

2. The function is neither restartable, nor resumable. That is, it may not be continued from immediate execution at all. An undamaged function or operator which is pendent upon a non-restartable function or operator may, however, be resumed by entering →0. Elements of ⎕*LC* corresponding

to such damaged functions or operators are set to 0. Lines reported by *)SI*, *)SINL*, and *)SIS* corresponding to such damaged functions or operators do not show line numbers.

The state indicator may be cleared with Abort statements, or with the system command *)RESET*.

SYNTAX ERROR

This report is given upon an attempt to either:

Execute an improper APL2 expression.

Execute ⎕*ES* 2 1

Execute ⎕*ES* 2 2

Execute ⎕*ES* 2 3

Execute ⎕*ES* 2 4

Examples:

```
      2×
SYNTAX ERROR
      2×
      ∧

      [1)
SYNTAX ERROR
      [1)
      ∧

      3←2
SYNTAX ERROR
      3←2
      ∧

      A←1
      (B←A)←7
SYNTAX ERROR
      (B←A)←7
        ∧
```

SYSTEM ERROR

This report is given when either:

> There is a fault in the internal operation of the APL2 system.
>
> There is damage detected in the active workspace.
>
> ⎕*ES* 1 2 is executed.

In either of the first two cases, the following action is taken:

1. Several lines of system related information is reported which may be helpful to the system manager in correcting the problem.

2. The damaged workspace is saved in a file in your private library which in many ways is like a normal stored workspace:

   a. It may be listed with the)*LIB* system command.

   b. It may be copied from with the)*COPY* or)*PCOPY* system command.

   c. It may be dropped from the library with the)*DROP* system command.

3. The active workspace is cleared.

The name used to store the damaged workspace is selected from the first unused of *DUMP*NNNN, where NNNN is a four digit number. For example, if a *SYSTEM ERROR* occurs, then the damaged workspace will be stored as *DUMP*0001 if that name is not already in use. If that name <u>is</u> already in use, then *DUMP*0002 will be a candidate, etc. This does not preclude a workspace being named *DUMP*NNNN, saved with the)*SAVE* command, and used normally.

SYSTEM LIMIT

This report is given upon an attempt to either:

> Use another slot in the system's internal symbol table but the table has reached its maximum size. This causes ⎕*ET* to be set to 1 4. The symbol table is automatically expanded whenever the system deems it necessary, but there is a max-

imum number of names that it can accommodate. The number of symbol slots currently in use in the symbol table is reported by the system command)*SYMBOLS*.

Share a variable when the shared variable facility is not in operation. This causes ⎕*ET* to be set to 1 5.

Share more variables, or use more shared variable storage than the maximum quota permitted by the shared variable interface portion of the APL2 system. This causes ⎕*ET* to be set to 1 6.

Use more shared variable storage than the capacity available at the time of need by the shared variable interface portion of the APL2 system. This causes ⎕*ET* to be set to 1 7.

Create an array of greater rank than the implementation limit (see "Appendix F. System Limitations" on page 327). This causes ⎕*ET* to be set to 1 8.

Create an array of more elements or a larger size than the implementation limit (see "Appendix F. System Limitations" on page 327). This causes ⎕*ET* to be set to 1 9.

Apply certain primitive functions to an array of greater depth, (or share such an array) than the implementation limit (see "Appendix F. System Limitations" on page 327). This causes ⎕*ET* to be set to 1 10.

Use a prompt in a prompt/response interaction which is longer than the maximum permitted by the terminal in use. This causes ⎕*ET* to be set to 1 11.

Use one of the library system commands)*COPY*,)*LIB*,)*LOAD*,)*PCOPY*, or)*SAVE* beyond a system limitation (such as available workspace or library size). This does not cause ⎕*ET* to be set.

Execute ⎕*ES* 1 4

Execute ⎕*ES* 1 5

Execute ⎕*ES* 1 6

Execute ⎕*ES* 1 7

Execute ⎕*ES* 1 8

Execute ⎕*ES* 1 9

Execute ⎕*ES* 1 10

Execute ⎕*ES* 1 11

Refer to the description of ⎕*ET* on page 208 for more details.

VALENCE ERROR

This report is given upon an attempt to either:

> Execute a strictly monadic function with both a left and a right argument.
>
> Execute a strictly dyadic primitive function (one which does not have a monadic definition) without a left argument.
>
> Execute ⎕*ES* 5 1

Examples:

```
      0⌻1 2 3
VALENCE ERROR
      0⌻1 2 3
      ^^

    ∇ Z←F R
[1]   Z←⎕NC 'R'
    ∇

      1 F 2
VALENCE ERROR
      1 F 2
      ^ ^

      ≤1
VALENCE ERROR
      ≤1
      ^^
```

VALUE ERROR

This report is given upon an attempt to either:

> Use a name which is not defined.
>
> Use a value from a function or operator which does not return a value.

Use a numeric constant whose value is too large (like 1*E*999) or too small (like ¯1*E*999) for the system implementation. (Infinitesimal values (like 1*E*¯999) will be converted to zero.)

Use a constant whose defining string is too long for the system implementation.

Execute ⎕*ES* 3 1

Execute ⎕*ES* 3 2

Examples:

```
      NOTHING
VALUE ERROR
      NOTHING
      ∧

    ∇F
[1]  1
    ∇

      ρF
1
VALUE ERROR
      ρF
      ∧

      1E9999
VALUE ERROR
      1E9999
      ∧
```

WS FULL

This report is given upon an attempt to either:

Execute any operation (including certain system commands) which requires more storage than is currently available.

Execute ⎕*ES* 1 3

In the first case, the remedy may require:

1. resetting the state indicator,

2. erasing needless objects,

3. revising calculations to save space.

Example:

```
      (10⍴10)⍴⍳10
WS FULL
      (10⍴10)⍴⍳10
      ^
```

WS LOCKED

This report is given upon an attempt issue one of the system commands *)COPY*, *)LOAD*, or *)PCOPY* when the stored workspace has a different password or key than is given in the command.

WS NOT FOUND

This report is given upon an attempt to issue one of the system commands *)COPY*, *)DROP*, *)LOAD*, or *)PCOPY* when the specified stored workspace does not exist.

⎕*CT ERROR*

This report is given upon an attempt to either:

> Execute a primitive function which uses ⎕*CT* as an implicit argument when ⎕*CT* has an inappropriate value or no value.
>
> Execute ⎕*ES* 4 3

Examples:

```
      ⎕CT←'X'
      1=2
⎕CT ERROR
      1=2
      ^^
```

```
    ∇ F;⎕CT
[1]   1=2
    ∇

      F
⎕CT ERROR
F[1] 1=2
     ^^
```

⎕*FC ERROR*

This report is given upon an attempt to either:

> Execute a primitive function which uses ⎕*FC* as an implicit argument when ⎕*FC* has an inappropriate value or no value.
>
> Display a complex number (which uses ⎕*FC* as an implicit argument) when ⎕*FC* has an inappropriate value or no value.
>
> Execute ⎕*ES* 4 4

Examples:

```
      ⎕FC←0
      ⍕0J1
⎕FC ERROR
      ⍕0J1
      ^^

    ∇ F;⎕FC
[1]   ⍕0J1
    ∇

      F
⎕FC ERROR
F[1] ⍕0J1
     ^^
```

⎕*IO ERROR*

This report is given upon an attempt to either:

> Execute a primitive function which uses ⎕*IO* as an implicit argument when ⎕*IO* has an inappropriate value or no value.

Execute ⎕ES 4 2

Examples:

```
      ⎕IO←'X'
      ⍳3
⎕IO ERROR
      ⍳3
      ∧∧

    ∇ F;⎕IO
[1]   ⍳3
    ∇

      F
⎕IO ERROR
F[1] ⍳3
     ∧∧
```

⎕MD ERROR

This report is given upon an attempt to either:

Execute a primitive function which uses ⎕MD as an implicit argument when ⎕MD has an inappropriate value or no value.

Execute ⎕ES 4 6

Examples:

```
      ⎕MD←'X'
      ⌹2 2⍴1 0 0 1
⎕MD ERROR
      ⌹2 2⍴1 0 0 1
      ∧∧

    ∇ F;⎕MD
[1]   ⌹2 2⍴1 0 0 1
    ∇

      F
⎕MD ERROR
F[1] ⌹2 2⍴1 0 0 1
     ∧∧
```

⎕*PP ERROR*

This report is given upon an attempt to either:

Execute a primitive function which uses ⎕*PP* as an implicit argument when ⎕*PP* has an inappropriate value or no value.

Display an array (which uses ⎕*PP* as an implicit argument) when ⎕*PP* has an inappropriate value or no value.

Execute ⎕*ES* 4 1

Examples:

```
      ⎕PP←'X'
      3.5
⎕PP ERROR
      3.5
      ∧

    ∇ F;⎕PP
[1]   3.5
    ∇

      F
⎕PP ERROR
F[1] 3.5
     ∧
```

⎕*PR ERROR*

This report is given upon an attempt to either:

Perform a prompt/response interaction when ⎕*PR* has an inappropriate value or no value.

Execute ⎕*ES* 4 7

Example:

```
     ∇ Z←F R;⎕PR
[1]    ⍞←R
[2]    Z←⍞
     ∇

       F '?'
?
⎕PR ERROR
F[2] Z←⍞
       ∧
```

⎕RL ERROR

This report is given upon an attempt to either:

Execute a primitive function which uses ⎕RL as an implicit argument when ⎕RL has an inappropriate value or no value.

Execute ⎕ES 4 5

Examples:

```
       ⎕RL←'X'
       ?2
⎕RL ERROR
       ?2
       ∧∧

     ∇ F;⎕RL
[1]    ?2
     ∇

       F
⎕RL ERROR
F[1] ?2
     ∧∧
```

Functions and operators may be defined with the system function Fix (⎕*FX*), or with the system editor (see "The APL2 Default Editor" on page 291 or "The APL2 Extended Editor" on page 295). The first line (line 0) of a defined function or operator is called the header. The remaining lines are called the body. Certain attributes of a defined function or operator are declared explicitly in the header and body.

VALENCE

There are three independent valences of a defined function or operator:

1. whether or not it produces an explicit result (0 or 1)

2. its function valence (0, 1, or 2)

3. its operator valence (0, 1, or 2)

The valences are defined in the header (line 0) of a defined function or operator. There are 18 combinations of these valences, but not all are valid. Figure 16 on page 276 shows the combinations. Entries in the table that show parentheses in the header are called defined operators (name class 4). Entries in the table that do not show parentheses in the header are called defined functions (name class 3).

The valences of a defined function or operator may be determined by inspecting its header (line 0), or from the dyadic system function Attributes (⎕*AT*). 1 ⎕*AT R* returns the explicit result, the function valence, and the operator valence (in that order).

A defined operator is specified by giving the definition of the derived function (see "Defined Operators" on page 21). The function valence of an operator is the valence of the derived function. An operator is defined by enclosing its name and its operands within parentheses in the header. The argument(s) of the derived function of an operator are outside the parentheses.

Defined functions and operators with function valence 2 may be called monadically (without a left argument). In such a case, the left argument will not have a value during execution, and its name class will be 0.

Expl. Res.	Oper. Val.	Function Valence		
		0	1	2
0	0	P	P R	L P R
0	1	invalid	(F P) R	L (F P) R
0	2	invalid	(F P G) R	L (F P G) R
1	0	Z←P	Z←P R	Z←L P R
1	1	invalid	Z←(F P) R	Z←L (F P) R
1	2	invalid	Z←(F P G) R	Z←L (F P G) R
Notes: P is the name of the defined function or operator. L is the left argument of the (derived) function. R is the right argument of the (derived) function. F is the left operand of the operator. G is the right operand of the operator. Z is the explicit result.				

Figure 16. Headers of Functions and Operators

LOCAL NAMES

The following names, if present, are local to a defined function or operator:

1. the result

2. the argument(s)

3. the operand(s) of an operator

4. additional names in the header after a semicolon (separated by a space)

5. labels (see the Branch statement on page 148 and "System Labels" on page 227)

The name of the defined function or operator is not local to itself, unless it is explicitly listed in the header after a semicolon. This permits recursive definitions.

During execution of a defined function or operator a local name will temporarily exclude from use a global object of the same name. This is called localization or shadowing. A value or meaning given to a local name will persist only for the duration of execution of the defined function or operator (including any time that it is suspended or pendent). Names which are not local are said to be global.

Example of a global variable:

```
     ∇ F
[1]   V←1
[2]   V
     ∇

        V←0
        F
1
        V
1
```

Example of a local variable:

```
     ∇ F;V
[1]   V←1
[2]   V
     ∇

        V←0
        F
1
        V
0
```

EXECUTION PROPERTIES

A defined function or operator may have four independent execution properties:

1. It cannot be displayed or edited, and its canonical representation is a matrix with shape 0 0.

2. It cannot be suspended, just as primitive functions cannot.

3. Weak interrupts will be ignored during its execution.

4. Any non-resource error within its scope will be converted into a *DOMAIN ERROR* (that is, *INTERRUPT*, *WS FULL*, and *SYSTEM LIMIT* are excluded from conversion).

The default function or operator definition (as provided by the system editors) is to have none of these properties. Each property may be set independently with the dyadic system function Fix (⎕*FX*), which is described on page 187. The system editors (∇) will not remove any execution properties if they have been previously assigned.

Execution properties are imposed through defined function and operator calls. For example, if function *F* has the non-displayable property, and it calls function *G*, then the non-displayable property will be imposed on function *G* whether or not *G* itself has the property. In this way, if a locked function calls an unlocked function, the unlocked function will behave as if it were locked.

Example of default execution properties:

```
      )CLEAR
CLEAR WS
      0 0 0 0 ⎕FX 'F' '1 2+3 4 5'
F
      F
LENGTH ERROR
F[1] 1 2+3 4 5
     ∧  ∧

      )SIS
F[1] 1 2+3 4 5
     ∧  ∧
```

Example of the non-displayable execution property:

```
      )CLEAR
CLEAR WS
      1 0 0 0 ⎕FX 'F' '1 2+3 4 5'
F
      F
LENGTH ERROR
F[1]

      )SIS
F[1]
```

Example of the non-suspendable execution property:

```
      )CLEAR
CLEAR WS
      0 1 0 0 ⎕FX 'F' '1 2+3 4 5'
F
      F
LENGTH ERROR
F[1] 1 2+3 4 5
     ∧  ∧

      )SIS
```

Example of the error conversion execution property:

```
      )CLEAR
CLEAR WS
      0 0 0 1 ⎕FX 'F' '1 2+3 4 5'
F
      F
DOMAIN ERROR
F[1] 1 2+3 4 5
     ∧  ∧

      )SIS
F[1] 1 2+3 4 5
     ∧  ∧
```

Having all four execution properties set is the same as being locked.

There are two facilities useful in analyzing the behavior of defined functions and operators, particularly during their design. They are trace control and stop control.

TRACE CONTROL

A trace is an automatic display of information generated by the execution of each selected line of a defined function or operator. When a statement is traced, the following information is displayed whenever the statement is executed:

1. the function or operator name

2. the line number (in brackets)

3. the final array value (or branch) produced by that statement

The trace control for a defined function or operator is designated by prefixing *T∆* to its name. For example, a trace may be set on lines 1, 3, and 6 of a defined function named *FN* by executing:

```
      T∆FN ← 1 3 6
```

A trace may be set on all lines of a defined operator named *OPR* (assuming that it has no more than 1000 lines) by executing:

```
      T∆OPR ← ⍳1000
```

Trace controls may be both set and referenced. A reference to a trace control vector returns only valid line numbers (in increasing order) upon which a trace has been set. Trace controls may be not be selectively specified.

Settings of Trace controls are not relocated as a result of line insertion or deletion by the system editor.

Example:

```
    ∇ Z←FACTORIAL R
[1]    Z←R⌈1
[2]   LOOP:R←R-1
[3]    →(R≤1)/0
[4]    Z←Z×R
[5]    →LOOP
    ∇
```

```
      TΔFACTORIAL ← ⍳5

      FACTORIAL 4
FACTORIAL[1] 4
FACTORIAL[2] 3
FACTORIAL[3] →4
FACTORIAL[3]
FACTORIAL[4] 12
FACTORIAL[5] →2
FACTORIAL[2] 2
FACTORIAL[3] →4
FACTORIAL[3]
FACTORIAL[4] 24
FACTORIAL[5] →2
FACTORIAL[2] 1
FACTORIAL[3] →0
24
```

More examples of Trace Control can be found in "System Labels" on page 227.

Names beginning with *TΔ* may not be used for any purpose other than trace control.

STOP CONTROL

A defined function or operator can be made to stop before a selected line is executed. This may be useful in analyzing things such as local variables. When a statement is assigned a stop control, execution stops just before the statement is to be executed, and the following information is displayed:

1. the function or operator name

2. the line number (in brackets)

Execution may be resumed by entering a branch statement.

The stop control for a defined function or operator is designated by prefixing *SΔ* to its name. For example, a stop may be set on lines 1, 3, and 6 of a defined function named *FN* by executing:

```
      SΔFN ← 1 3 6
```

A stop may be set on all lines of a defined operator named *OPR* (assuming that it has no more than 1000 lines) by executing:

```
      SΔOPR ← ⍳1000
```

Stop controls may be both set and referenced. A reference to a stop control vector returns only valid line numbers (in increasing order) upon which a stop has been set. Stop controls may be not be selectively specified.

Settings of Stop controls are not relocated as a result of line insertion or deletion by the system editor.

Example:

```
    ∇ Z←FACTORIAL R
[1]    Z←R⌈1
[2]   LOOP:R←R-1
[3]    →(R≤1)/0
[4]    Z←Z×R
[5]    →LOOP
    ∇

      SΔFACTORIAL ← 5

      FACTORIAL 4
FACTORIAL[5]
      R
3
      Z
12
      →5
FACTORIAL[5]
      R
2
      Z
24
      →5
24
```

Names beginning with *S*Δ may not be used for any purpose other than stop control.

Figure 17 displays the set of characters in APL2. The positions of the characters shown in the table correspond to 16 16⍴⎕*AV* (the Atomic Vector system variable), and also indicate their hexadecimal representation in EBCDIC code. The hexadecimal representation *X* of a character gives its row and column in the table. A corresponding index to ⎕*AV* can be obtained by the expression 1+16⊥¯1+'0123456789*ABCDEF*'⍳*X*.

	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
00																	00
10																	10
20																	20
30																	30
40		*A̲*	*B̲*	*C̲*	*D̲*	*E̲*	*F̲*	*G̲*	*H̲*	*I̲*	¢	.	<	(	+	|	40
50	&	*J̲*	*K̲*	*L̲*	*M̲*	*N̲*	*O̲*	*P̲*	*Q̲*	*R̲*	!	$	*	)	;	¬	50
60	-	/	*S̲*	*T̲*	*U̲*	*V̲*	*W̲*	*X̲*	*Y̲*	*Z̲*	¦	,	%	_	>	?	60
70		∧	¨	⌻	⍸	⍷			∨	`	:	#	@	'	=	"	70
80	~	a	b	c	d	e	f	g	h	i	↑	↓	≤	⌈	⌊	→	80
90	⎕	j	k	l	m	n	o	p	q	r	⊃	⊂		○		←	90
A0	¯	~	s	t	u	v	w	x	y	z	∩	∪	⊥	[	≥	∘	A0
B0	⍺	∊	⍳	⍴	⍵		×	\	÷		∇	∆	⊤	]	≠	∣	B0
C0	{	*A*	*B*	*C*	*D*	*E*	*F*	*G*	*H*	*I*	⍲	⍱	⌷	⌽	⍂	⍉	C0
D0	}	*J*	*K*	*L*	*M*	*N*	*O*	*P*	*Q*	*R*	⌶	!	⍒	⍋	⍞	⍝	D0
E0	\	≡	*S*	*T*	*U*	*V*	*W*	*X*	*Y*	*Z*	⌿	⍀	∵	⊖	⍠	▼	E0
F0	0	1	2	3	4	5	6	7	8	9		⍫	⍙	⍟	⍎		F0
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	

Figure 17. The APL2 Character Set

The italic upper case letters in the character set are displayed as block upper case letters on many terminals and printers.

Figure 18 on page 286 shows those characters, and their names, which have meaning in the APL2 Language. The names of the characters do not necessarily indicate the operations that they represent. Included in the table are the pages in this manual where the main descriptions begin for the use of each symbol in APL2.

The alphabetic characters are:

ABCDEFGHIJKLMNOPQRSTUVWXYZ
A̲B̲C̲D̲E̲F̲G̲H̲I̲J̲K̲L̲M̲N̲O̲P̲Q̲R̲S̲T̲U̲V̲W̲X̲Y̲Z̲
∆∆̲

¨	dieresis	156	157	⊂	left shoe	46	
¯	overbar	5		⊃	right shoe	56	114
<	less	83		∩	cap	61	
≤	not greater	89		∪	cup	54	
=	equal	82		⊥	base	132	
≥	not less	90		⊤	top	132	
>	greater	83		|	stile	39	99
≠	not equal	88		;	semicolon	146	
∨	or	91		:	colon	17	148
∧	and	79		,	comma	49	95
-	bar	40	93	.	dot	178	176
÷	divide	41	81	\	backslash	107	165
+	plus	36	78	/	slash	115	160
×	times	36	87	⍱	nor	88	
?	query	42	122	⍲	nand	87	
⍵	omega			⍒	del stile	57	128
∊	epsilon	42	131	⍋	delta stile	59	129
⍴	rho	74	99	⌽	circle stile	52	100
~	tilde	40	120	⍉	circle bksl.	53	102
↑	up arrow	118		⊖	circle bar	52	100
↓	down arrow	105		⍟	circle star	40	84
⍳	iota	61	131	⌶	I-beam		
○	circle	41	80	⍫	del tilde	291	295
⋆	star	37	91	⍎	base jot	69	
→	right arrow	148		⍕	top jot	70	138
←	left arrow	6	22	⍀	backslash bar	107	165
⍺	alpha			⌿	slash bar	115	160
⌈	up stile	38	86	⍝	cap jot	17	
⌊	down stile	35	85	⍞	quote quad	204	
_	underbar	5		!	quote dot	37	79
∇	del	291	295	⌹	domino	65	133
∆	delta	5		⍙	delta underbar	5	
∘	jot	176		⍂	quad bksl.	64	
'	quote	5		⌻	quad jot	67	
⎕	quad	213		⌷	squad	60	109
(	left paren	6	15	⍢	dotted del		
)	right paren	6	15	≡	equal underbar	68	137
[	left bracket	146	168	⍷	epsilon under.	122	
]	right bracket	146	172	⍸	iota underbar	125	
	blank	6	15				

Figure 18. Names of APL2 Characters

The alphanumeric characters include the alphabetic characters, and also:

0123456789_¯

The blank is encoded as X'40', which is ⎕*AV*[65].

The following are special characters valid in APL2 functions or character constants:

X'50'	⎕AV[81]	&	ampersand
X'6C'	⎕AV[109]	%	percent

The following are national use characters and may have different graphics in different countries:

X'4A'	⎕AV[75]	¢	cent
X'4F'	⎕AV[80]	|	vertical bar
X'5A'	⎕AV[91]	!	exclamation
X'5B'	⎕AV[92]	$	dollar
X'5F'	⎕AV[96]	¬	not
X'6A'	⎕AV[107]	¦	split bar
X'79'	⎕AV[122]	`	accent
X'7B'	⎕AV[124]	#	pound
X'7C'	⎕AV[125]	@	at
X'7F'	⎕AV[128]	"	double quote
X'A1'	⎕AV[162]	~	tilde (national)
X'C0'	⎕AV[193]	{	left brace
X'D0'	⎕AV[209]	}	right brace
X'E0'	⎕AV[225]	\	backslash (national)

The following are non-printable terminal control characters not valid in APL2 functions or character constants:

X'15'	⎕AV[22]	new line (carriage return)
X'16'	⎕AV[23]	backspace
X'25'	⎕AV[38]	line feed

The following are characters reserved for future APL2 use (not currently valid in APL2 functions or character constants):

X'70'	⎕AV[113]
X'76'	⎕AV[119]
X'77'	⎕AV[120]
X'9C'	⎕AV[157]
X'9E'	⎕AV[159]
X'B5'	⎕AV[182]
X'B9'	⎕AV[186]
X'FA'	⎕AV[251]

In addition to the reserved characters, X'FF' (⎕AV[256]), and X'00' through X'39' (40↑⎕AV) are not valid in APL2 functions or character constants.

The appearance of some characters may be very similar even though they are distinct:

X'80'	⎕*AV*[129]	~	APL tilde
X'A1'	⎕*AV*[162]	~	national tilde
X'B7'	⎕*AV*[184]	\	APL backslash
X'E0'	⎕*AV*[225]	\	national backslash
X'79'	⎕*AV*[122]	`	national accent
X'DB'	⎕*AV*[220]	!	APL quote dot
X'5A'	⎕*AV*[91]	!	national exclamation
X'BF'	⎕*AV*[192]	|	APL stile
X'4F'	⎕*AV*[80]	|	national vertical bar

Since some terminals may be unable to form certain characters, overstrike pair combinations are accepted for all of the compound APL2 characters, the underscored letters, the lower case letters, the special characters, and the national characters. Characters in a pair may occur in either order, with an intervening backspace. The overstrike pair combinations are shown in Figure 19 on page 289. Although the national use characters may have alternate graphics in different countries, they do not have alternate overstrike combinations.

A̲	A_	a	A¯	N̲	N_	n	N¯
B̲	B_	b	B¯	O̲	O_	o	O¯
C̲	C_	c	C¯	P̲	P_	p	P¯
D̲	D_	d	D¯	Q̲	Q_	q	Q¯
E̲	E_	e	E¯	R̲	R_	r	R¯
F̲	F_	f	F¯	S̲	S_	s	S¯
G̲	G_	g	G¯	T̲	T_	t	T¯
H̲	H_	h	H¯	U̲	U_	u	U¯
I̲	I_	i	I¯	V̲	V_	v	V¯
J̲	J_	j	J¯	W̲	W_	w	W¯
K̲	K_	k	K¯	X̲	X_	x	X¯
L̲	L_	l	L¯	Y̲	Y_	y	Y¯
M̲	M_	m	M¯	Z̲	Z_	z	Z¯

⍱	∨~	nor
⍲	∧~	nand
⍒	∇|	del stile
⍋	∆|	delta stile
⌽	○|	circle stile
⍉	○\	circle backslash
⊖	○-	circle bar
⍟	○⋆	circle star
⌶	⊥⊤	I-beam
⍫	∇~	del tilde
⍎	⊥∘	base jot
⍕	⊤∘	top jot
⍀	\-	backslash bar
⌿	/-	slash bar
⍝	∩∘	cap jot
⍞	'⎕	quote quad
!	'.	quote dot
⌹	⎕÷	domino
⍙	∆_	delta underbar
⍂	⎕\	quad backslash
⌻	⎕∘	quad jot
⌷	[ ]	squad
∵	¨.	dotted del
≡	=_	equal underbar
⍷	∊_	epsilon underbar
⍸	⍳_	iota underbar

&	∊|	ampersand
%	÷/	percent
¢	⊂|	cent
|	⊥|	vertical bar
!	'∘	exclamation
$	S/	dollar
¬	~/	not
¦	',	split bar
`	¯\	accent
#	N=	pound
@	Q∘	at
"	'¨	double quote
~	~|	tilde (national)
{	-(	left brace
}	-)	right brace
\	\\|	backslash (national)

Figure 19. Overstrike Combinations

The APL2 Default Editor will edit (the most local version of) defined functions and defined operators. It is entered with the ∇ or ⍫ command if the system command)*EDITOR* 1 has been issued, or if no)*EDITOR* command has been issued.

The ∇ command may only be used outside the editor to enter edit mode. It may take one of the following forms:

∇ header

Begin editing a new defined function or operator with the specified header (line zero).

∇ name

Begin editing the existing defined function or operator with the specified name.

∇ name command

Begin editing the existing defined function or operator with the specified name, and execute the specified first bracketed command.

∇ name command ∇

Begin editing the existing defined function or operator with the specified name, execute the specified bracketed command, and then leave edit mode, returning to immediate execution.

There are several commands available within edit mode in the default editor which display or change the object being edited. Some of the bracket command forms take line number specifications. Any line numbers may be fractional. In the following, L and M are single line numbers, and V is a line number list (possibly containing redundant blanks):

[L] text

Replace or insert the specified text line at the specified single line number L. The line number may be fractional. Trailing blanks in a comment will be deleted.

[⎕]

Display all of the object text lines.

[Δ]

Delete all of the object text lines.

[⎕V]

Display the object text line(s) mentioned in V, which may be in any order, and may contain repetitions.

[⎕L-M]

Display the object text lines in the interval from L to M, inclusive.

[⎕-L]

Display the object text lines from the beginning of the object to line L, inclusive.

[⎕L-]

Display the object text lines from line L to the end of the object, inclusive.

[ΔV]

Delete the object text line(s) mentioned in V, which may be in any order.

[ΔL-M]

Delete the object text lines in the interval from L to M, inclusive.

[Δ-L]

Delete the object text lines from the beginning of the object to line L, inclusive.

[ΔL-]

Delete the object text lines from line L to the end of the object.

(Note that if object lines are unintentionally deleted, the editing session can be aborted with the [→] command, and the original object will remain intact.)

[L⎕M]

Edit the single object text line numbered L at position M in a detailed manner (this is also called super-edit). The edit action depends on the type of terminal.

On a display terminal:

1. Display the text line in the input area.

2. If M is 0, then place the cursor just after the end of the line.

   If M is not zero, then place the cursor at position M of the line.

3. Accept input to change the line.

On a non-display terminal:

1. Display the text line.

2. If M is 0, then place the cursor just after the end of the line, and accept input to continue the line.

   If M is not zero, then place the cursor under position M of the line, and accept edit characters:

   / will delete the character above it.

   A digit will insert the specified number of spaces immediately to the left of the character above it.

   A letter will insert spaces in multiples of five (5 for *A*, 10 for *B*, 15 for *C*, etc).

3. If M was not zero, re-display the new line with the characters deleted and the spaces inserted. Then accept input to fill in the blanks or form overstrikes.

[→]

Abort editing the object without establishing it in the workspace, and return to APL2 immediate execution. (If → is entered on a line by itself, then this will be treated as if it were [→]).

∇

Establish the edited object in the workspace, cease editing the object, and return to APL2 immediate execution.

The defined function or operator is not established in the workspace until definition is closed with the ∇. If a function is renamed by editing its header, the old function is not expunged when the new one is established.

⍫

> Establish the edited object in the workspace, lock it, cease editing the object, and return to APL2 immediate execution.

While in edit mode, the system supplies prompts for new lines. Any input line which begins with a right parenthesis (which can be entered by deleting the prompt) will be executed as a system command. Any response to the system command is not treated as an edit command. Some system commands will cause the editing to be aborted.

Any input line which does not begin with one of the characters) → ∇ ⍫ or [is considered an APL2 statement. It will cause editing to be suspended, and the line to be evaluated in immediate execution in the active workspace. After the interruption, editing will be resumed.

A closing ∇ or ⍫ may optionally end any edit command (except a text line ending with a comment), or it may be on a line by itself. This means to execute the edit command, and then leave edit mode. If the ⍫ character is used instead of the ∇ character, then the editor action is the same, but the function will be locked.

The system commands)*SI*,)*SINL*, and)*SIS* will identify with the character ∇ the names of defined functions or operators that are suspended in editing.

The APL2 Extended Editor will edit (the most local version of) defined functions, defined operators, and character vectors or matrices. It is entered with the normal ∇ or ⍫ command if the system command)*EDITOR* 2 has been issued.

The APL2 Extended Editor operates in full screen mode if it is used from a display terminal. If the terminal is not a display terminal, or if the full screen processor is not available (or if it abends), then the editor will operate in non-full screen mode similar to the default APL2 editor, except that there will be no prompting for line numbers. All commands described here are available in either case.

If the ∇ or ⍫ command is issued within the editor, the display terminal screen is split vertically into partitions or segments, and multiple objects may be viewed and edited simultaneously. In such a case, a given edit command will affect only the object being edited in the screen segment where the command was entered.

The following discussion explains the details of the operation of the extended editor. A somewhat less detailed explanation is given in "Editing Hints" on page 308.

SCREEN PROCESSING

Lines may be typed onto any line of the screen. All lines on the screen are scanned during processing. Screen processing is performed from top to bottom. An updated screen (as determined by the input lines) is displayed after all input lines have been processed.

If no edit command which explicitly requests a display appears in the screen segment, then the default system response is to re-display that part of the object which is in that segment. This display begins with an object information line. Thus, the effect of many simultaneous alterations may be seen immediately.

If any edit command which explicitly requests a display does appear in the screen segment, then the system action is to not change the display except for honoring the request. All edit commands will still be processed, but the effects of commands which do not explicitly request displays may not be seen immediately. In such a case, if the ENTER key is used again without entering any other commands, then the effects of the previous commands can be seen.

Editor processing is started with either the ENTER key or a PF key. If a PF key is used, then screen processing is performed normally, and then the PF key's definition is executed as if it were the last line in the screen segment where the cursor was when the key was pressed. The PF key assignments are shown in Figure 20 on page 307.

If the PA2 key is used twice in succession, then the editor will be immediately aborted.

If an editing error occurs, screen processing will stop, the cursor will be placed at the beginning of the line in error, and the terminal alarm will sound. An error message will be placed on the top line of the screen segment where the error occurred, overwriting the object information line. Unprocessed commands below the line in error will remain on the screen so that they may be processed after the error has been corrected.

There are four classes of input lines that the editor recognizes:

1. special commands that affect the editing environment

2. text lines identified by preceding bracketed line numbers that display, alter, or insert object text

3. bracketed commands which display or change object text

4. unidentified lines (all lines not belonging to the first three classes)

The effect of an input line may depend upon where it is typed on the screen. If more than one object is being edited simultaneously, then an input line will affect only the object being edited in the screen segment where it was entered.

An input line may be typed over the object information line which the system gives at the top of each screen segment.

Blank lines are ignored by the editor during processing. System commands are not recognized by the editor.

SPECIAL COMMANDS

There are two special commands which affect the editing environment:

∇ name command

Begin editing the object with the given name. This command may be used either outside the editor to enter edit mode, or inside the editor to edit multiple objects simultaneously.

If the command is entered while in edit mode, the screen line where it was entered will begin a new object segment, and the previous segment will end on the previous line.

If the object is a new one, then it is assumed to be a defined function or operator, and the entire header (line 0) may be provided instead of only the name.

Optionally, the first input line may also be included with the ∇ command.

Optionally, a closing ∇ may be included last. This means to:

1. Open editing of the object.

2. Establish a temporary screen segment for the object.

3. Execute only the specified first input line, if it was included.

4. Establish the object in the workspace if it was changed.

5. Cease editing the object.

6. Release the temporary screen segment.

If the ∇ command is entered with a closing ∇ while in edit mode (∇*F*[⎕]∇ for example), then the temporary screen segment for this object (*F*) will not be cleared, but will be appended to the preceding segment as is. This may be helpful to insert the text of one object into another, when followed by appropriate line number modifications.

If the ∇ command is entered with a closing ∇ while not in edit mode, then the temporary screen segment for this object will be displayed at the terminal, and non-edit mode will be continued.

If the ⍫ character is used instead of the ∇ character, then the editor action is the same, but the function will be locked when the definition is closed.

∇

End editing the object in the segment where the command was typed. This command is distinguished from the preceding one because it has no arguments. It means to:

1. Establish the object being edited in the active workspace.

2. Cease editing the object.

3. Clear the screen segment for this object (replace it with blanks) if the ∇ command was entered on a line by itself.

4. Release the screen segment for this object:

   a. Expand the preceding screen segment. If there is no preceding segment, then

   b. Expand the succeeding segment. If there is no other segment, then

   c. Leave edit mode.

The ∇ command (without arguments) is initially assigned to PF3. It will close the segment in which the cursor is found.

A closing ∇ may optionally end any text input line, bracketed command, or an un-identified line, or it may be on a line by itself. The screen segment is cleared (step 3 above) only when the ∇ is entered on a line by itself. If the ∇ is found at the end of an input line, then the screen segment is released (as above) without clearing it.

If the ⍫ character is used instead of the ∇ character, then the editor action is the same, but the function will be locked.

EXPLICITLY NUMBERED TEXT LINES

An explicitly numbered text line is used for both text display and text input.

Object text lines are displayed on the screen preceded by a bracketed line number identification. Object text lines that are too long to fit on one line of the screen occupy multiple display lines on the screen, with continuation lines identified by empty brackets. Functions and operators will have non-comments and non-labeled lines indented one space. Trailing blanks in object text lines will be replaced with null characters on the screen, so that terminal insertion mode may be used.

The following are text input lines:

[L] text

Replace or insert the specified text line at the specified single line number L. The line number may be fractional.

In a function or operator, leading blanks in a text line will be deleted. In a character variable, the first blank after the closing bracket of the command (if it exists) will not be considered part of the text line. Trailing blanks in a text input line will not be significant, unless the entire line after the brackets is blank.

Text lines may begin with bracketed expressions. Only the first set of brackets identifies a text input line. Changing the line number of a text line makes a new copy of the line, and does not delete the old line.

A text line may not end with a ∇, because it would be interpreted as a closing ∇.

Note that because the screen is processed from top to bottom, the last of conflicting text input lines will be the effective one.

[ ] text

Continue the last object text line preceding (above) this one in the screen segment.

The first two blanks after the closing bracket of the command (if they exist) will not be considered part of the text line. Trailing blanks in a text continuation line will not be significant, unless the entire line after the brackets is blank.

A text continuation line need not be typed on the screen immediately below the line it continues. Intervening blank lines are permitted. An intervening bracket command, however, will cause a text continuation line to be ignored. If a numbered text line does not appear in the screen segment above it, then it will be treated as a continuation of the last text line in the object.

A text line may not end with a ∇, because it would be interpreted as a closing ∇.

A closing ∇ may optionally end any explicitly numbered text line. This means to close the screen segment after processing the line (as described in "Special Commands" on page 296), but suppress clearing the screen segment. Any commands appearing below a closing ∇ will not be processed. The last character of text in a text line may not be a ∇, because it will be treated a a closing ∇.

IMPLICITLY NUMBERED TEXT LINES

Text input may be entered without the preceding explicit line number identification within brackets. An input line in a normal screen segment (not an execute segment) which does not begin with ∇ or [may be accepted as new text. It will be given an implicit line number such that the line is inserted into the object immediately after the last numbered text line appearing above it in the screen segment. Implicitly numbered text lines will be re-displayed with their assigned line numbers.

An implicitly numbered text line need not be typed on the screen immediately below the line it follows. Intervening blank lines are permitted. An intervening bracket command, however, will cause an implicitly numbered text line to be inserted after the last text line in the object. Also, if a numbered text line does not appear in the screen segment above an implicitly numbered text line, then it will be inserted after the last text line in the object.

If an implicit text line and a subsequent command requesting a display are entered in the same screen segment, then the re-display of the new text line with its assigned number will be inhibited. In such a case, if the implicit line is re-processed, then it will be inserted into the object again. Therefore, simultaneous display requests and implicit line numbering is not recommended.

A closing ∇ may optionally end any implicitly numbered text line. This means to close the screen segment after processing the line (as described in "Special Commands" on page 296), but suppress clearing the screen segment. Any commands appearing below a closing ∇ will not be processed. The last character of text in a text line may not be a ∇, because it will be treated a a closing ∇. Implicitly numbered text lines may not begin with a left bracket ([).

PRIMARY BRACKET COMMANDS

There are five forms of primary bracket commands which display or change an object being edited. If multiple objects are being edited, the affected one is the one in the screen segment where the command was typed.

The scope of a primary bracket command is indicated by its form. Four of the five bracket command forms take line number specifications. They are distinguished by the presence and position of a bar (-). In the following description, L and M are single line numbers, and V is a list of one or more line numbers (possibly containing redundant blanks). Any of the line numbers may be fractional. Line numbers may not appear on the left side of a command. Any text (except a closing ∇) which appears to the right of the closing right bracket is ignored.

[command]

Apply the command to all of the text lines in the object.

[command V]

Apply the command to all of the text lines mentioned in V, which may be in any order, and may contain repetitions.

[command L-M]

Apply the command to all of the text lines in the interval from L to M, inclusive.

[command -L]

Apply the command to all of the text lines in the interval from the beginning of the object to L, inclusive.

[command L-]

Apply the command to all of the text lines in the interval from L to the end of the object, inclusive.

Any of the four primary bracket command forms above may be used with any of four "command" types below: display, delete, locate, and change.

⎕

Display the specified text lines (or as many as can be displayed in the screen segment), beginning on the line where the command was typed. If the requested display does not extend to the bottom of the screen segment, then the display will be padded at the bottom with blank lines to the end of the segment. The display may be truncated by another display or result-producing command. This command requests a display, so it inhibits a re-display of the text lines in the segment.

Examples:

[⎕] will display all object text lines (that will fit in the segment).

[⎕2 7] will display the object text lines numbered 2 and 7.

[⎕2-7] will display the object text lines numbered 2 through 7.

[⎕-2] will display the object text lines numbered 2 or less.

[⎕2-] will display the object text lines numbered 2 or greater.

Note that because the screen is processed from top to bottom, a display command will not reflect alterations to the object which appear on the screen below the display command. Therefore, to avoid possible confusion, a display command should not normally be entered on the screen where there are other commands below it.

Note that because the display may be truncated by another display or result-producing command, and because screen displays are used as input lines during subsequent processing, it is possible to lose continuation text lines by the truncation of displays.

Δ

Delete the specified text lines. Text lines are never deleted except by this command. Erasing a text line or changing its line number on the screen does not delete it.

Examples:

[Δ] will delete all object text lines.

[Δ2 7] will delete the object text lines numbered 2 and 7.

[Δ2-7] will delete the object text lines numbered 2 through 7.

[Δ-2] will delete the object text lines numbered 2 or less.

[Δ2-] will delete the object text lines numbered 2 or greater.

Note that because the screen is processed from top to bottom, a delete command that follows a text input line with the same number renders the text input line ineffective. Also, a text input line that follows a delete command may replace a deleted line.

Note that because text appearing to the right of the closing bracket is ignored, a text line displayed on the screen may be deleted by inserting a ∆ after the left bracket and before the line number. This is the recommended way to delete a line.

(Note that if object lines are unintentionally deleted, the editing session can be aborted with the [→] command, and the original object will remain intact.)

/string/ options

Locate and display the specified text lines which contain the specified string of characters. This command requests a display, so it inhibits a re-display of the text lines in the segment.

The character delimiter / which identifies the string may be any non-alphanumeric character not occurring in the string, and not any of .]→↓↑⍳?⍝⎕∆∇-. If there are no options, then the closing delimiter / may be elided.

The options may include blanks, and the characters N and ¨. Blanks in the options are ignored. If the options include the letter N, then the search will be performed for APL2 names only. An APL2 name, in this context, is a string which is neither preceded nor succeeded immediately by an alphanumeric character, or by ⎕ or ⍞. This includes names in comments and character constants (quoted strings). The ¨ option is permitted, but has no effect with this command.

Examples of locate:

[/*AT*/] will locate and display all the object text lines which contain the character string *AT*.

[/*AT*/ 2 3 7] will locate and display the object text lines numbered 2, 3, or 7 if any of them contains the character string *AT*.

[/*AT*/ 2-7] will locate and display the object text lines in the interval numbered 2 through 7 which contain the character string *AT*.

[/*AT*/ -2] will locate and display the object text lines numbered 2 or less which contain the character string *AT*.

[*/AT/* 2-] will locate and display the object text lines numbered 2 or greater which contain the character string *AT*.

[*/AT/ N*] will locate and display all the object text lines which contain the APL2 name *AT*. This will not locate uses which are not complete names, like *THAT*, *AT*13, or ⎕*AT*, because *AT* is only part of a larger name or word.

/old/new/ options

Locate the specified text lines which contain the specified old string of characters, and replace each indicated occurrence (see the ¨ option, below) of the old string with the new string.

The character delimiter / which identifies the string may be any non-alphanumeric character not occurring in the string, and not any of .]→↓↑⍳?⍝⎕∆∇-.

The options may include blanks, and the characters N and ¨. Blanks in the options are ignored. If the options include the letter N, then the search and changes are performed for old strings that are APL2 names only. An APL2 name is a string which is neither preceded nor succeeded immediately by an alphanumeric character, or by ⎕ or ⍞. This includes names in comments and character constants (quoted strings).

If the options include the character ¨, then each non-overlapping occurrence on the affected lines will be changed. Otherwise, only the first occurrence on each line will be changed.

Examples of change:

[*/AT/ROW/*] will change the character string *AT* to *ROW* the first time it occurs in any object text line.

[*/AT/ROW/* ¨] will change the character string *AT* to *ROW* every time it occurs in any object text line.

[*/AT/ROW/* ¨ 2 7] will change the character string *AT* to *ROW* every time it occurs in object text line 2 or 7.

[*/AT/ROW/* ¨ 2-7] will change the character string *AT* to *ROW* every time it occurs in any object text line numbered 2 through 7.

[*/AT/ROW/ N*] will change the APL2 name *AT* to *ROW* the first time it occurs in any object text line.

This will not change uses which are not names, like *THAT*, *AT*13, or ⎕*AT*.

[/*AT*/*ROW*/ ¨ *N*] will change the APL2 name *AT* to *ROW* every time it occurs in any object text line. This will not change uses which are not names, like *THAT*, *AT*13, or ⎕*AT*.

[/*AT*/*ROW*/ ¨ *N* 2-7] will change the APL2 name *AT* to *ROW* every time it occurs in any object text line numbered 2 through 7. This will not change uses which are not names, like *THAT*, *AT*13, or ⎕*AT*.

MISCELLANEOUS BRACKET COMMANDS

There are several miscellaneous bracket commands which may or may not not alter an object being edited, and will generally change the display. Most of them are especially useful when called from PF keys, and are assigned initially to a PF key (see Figure 20 on page 307).

[↓]

Scroll down (forward) through the object and re-display the screen segment, so that the text line that the cursor is on (or the last text line before the cursor) is displayed at the top of the screen segment. The segment display will not end with part of a continued text line.

This command is particularly useful on a PF key. It is initially assigned to PF8. The cursor is not moved after the scrolling, so that, for example, repeated scrolling down by five lines can be done by placing the cursor on the sixth line from the top of the screen segment and repeatedly using PF8.

[↑]

Scroll up (backward) through the object and re-display the screen segment, so that the text line that the cursor is on (or the last text line before the cursor) is displayed at the bottom of the screen segment. The segment display will not begin with a continuation text line.

This command is particularly useful on a PF key. It is initially assigned to PF7. The cursor is not moved after the scrolling, so that, for example, repeated scrolling up by five lines can be done by placing the cursor on the sixth line from the bottom of the screen segment and repeatedly using PF7.

[⍳]

Renumber all object text lines in the segment where the command was typed with consecutive integers, and then re-display the object beginning at the top of the screen segment area for this object. This command is initially assigned to PF9.

[?]

Display the current PF key assignments as a comment on the screen line where the command was entered. The PF key assignments are shown in Figure 20 on page 307. This command is initially assigned to PF1. This command requests a display, so it inhibits a re-display of the text lines in the segment.

Since a PF key's definition is executed as if it were the last line in the screen segment where the cursor was, PF1 may be used to identify the bottom line of a screen segment.

[∇]

Establish the object being edited in the active workspace. Do not release the screen segment, but continue editing the object. This command is initially assigned to PF6.

[→]

Abort editing the object being edited in the screen segment where the command was typed. Do not establish it in the workspace, but clear the screen segment, and release it to the adjacent segment as with the ∇ command.

[⍝]

Ignore this screen line (except for a closing ∇). It is a comment only. The editor places such an information comment line at the top of each screen segment, but this line may be over-written with an input line.

Any text (except a closing ∇) which appears to the right of the closing right bracket is ignored.

IMMEDIATE EXECUTION SEGMENTS

A special screen segment may be created under the name ⍎ with the command ∇⍎. Special rules apply to such a segment:

1 [?]	2	3 ∇
4	5	6 [∇]
7 [↑]	8 [↓]	9 [⍳]
10	11	12

Figure 20. PF Key Assignments for the Extended Editor

Immediate execution is permitted.

Implicitly numbered text input lines are not permitted.

Bracket commands and explicitly numbered text lines are permitted, but any resulting text can only be displayed while the segment exists, and cannot be established in the workspace. They may, however, be appended to an adjacent screen segment with a closing ∇ at the end of a numbered text line.

Any line in an immediate execution screen segment which does not begin with the characters ∇ or [is considered an APL2 statement, and will be executed in the active workspace if the cursor was on that line when processing began. Such a line will be indented six spaces in the display after processing.

If the line has a result, then the result will be displayed on the screen, beginning just under the line where the statement was typed. If the display of the result does not extend to the bottom of the screen segment, then the display will be padded at the bottom with blank lines to the end of the display.

If execution of the statement results in an APL2 error, then the error report will be displayed on the screen beginning just under the line where the statement was typed. Errors resulting from immediate execution statements are not considered editing errors, and they will not stop screen processing.

Branch statements may not be executed in an execute segment. The statement → is equivalent to [→] if the cursor was on that line when processing began.

The system commands *)SI*, *)SINL*, and *)SIS* will identify with the character ∇ the names of defined functions or operators that are suspended in editing.

EDITING HINTS

In the following discussions, mention is made of editing functions only. Defined functions, defined operators, and (character vector or matrix) variables may all be edited, and the discussions apply to any of these objects.

Hints for creating a new function:

1. Because of implicit line numbering, it is only necessary to type the text of lines on the screen. They may be typed in any order, and with intervening blank lines if desired. They are put into the function in the order they appear on the screen. The system will insert appropriate line numbers whenever the ENTER key is used.

2. If it is necessary to enter a text line that is longer than the width of the screen, type its continuation on the screen line immediately below the first part, and preceded with [ ].

3. When the screen is full, you may leave the cursor near the bottom of the screen and use the PF8 key to scroll down.

Hints for examining an existing function:

1. The PF7 and the PF8 key may be used in conjunction with the cursor position to scroll through the function.

2. If it is desired to perform a search for a name or a character string, a locate command (like [/⍝RESULT/] can typed directly over the top text line in the screen segment. That way, the located text lines will be displayed beginning at the top of the segment, so there will be plenty of room for them.

3. After such a locate command has been processed, the section of the function that begins with one of the given lines can be conveniently displayed. Simply erase all the text lines above the desired one (with the ERASE EOF key), and then use the ENTER key. The function will be re-displayed, beginning with the top text line showing in the segment.

4. A passive edit screen segment may be aborted by typing a right arrow (→) over one of the line numbers (or the ⍝) in brackets in the segment.

Hints for changing an existing function:

1. To perform minor changes to text lines, just display the appropriate part of the function, and type the changes over any text line on the screen. Characters can be removed from text lines with the DEL key. Characters can be added to short text lines by using the INS MODE key.

2. If it is necessary to insert characters in the middle of a long text line which has a continuation, the best method is to use the change command for the single line (like [/*VAR*/*VARIABLE*/ 17]) Type the change command below the last continuation of the text line to be changed, and then use the ENTER key. Commands may be typed over any text line.

3. The best way to delete a text line from the function is to display it for verification, and then insert a ∆ after the [and before the line number.

4. Text lines may be inserted by using either implicit or explicit line numbers. If implicit line numbers are used, just type the new line under the one it is to follow. If explicit line numbers are used, the line may be typed anywhere in the screen segment. When the ENTER key is used, the lines will be re-displayed with line numbers, and in their proper order.

   Existing text lines which are not continued may be freely overwritten anywhere on the screen. Care must be taken to erase any continuation lines if part of a continued text line is overwritten. The existing text line will not be affected by overwriting it.

5. An exact copy can be made of a text line by changing only its line number. The original text line will be unaffected. When the ENTER key is used, the lines will be re-displayed in their proper order.

6. An similar copy of a text line can be made by changing both its line number and part of its text. The original text line will be unaffected. When the ENTER key is used, the lines will be re-displayed in their proper order.

7. A text line can be appended to the one showing above it by blanking out its line number. When the ENTER key is used, the lines will be re-displayed.

8. Multiple commands and text changes may be typed on the screen before the ENTER key is used and the modified function is re-displayed. After commands and text changes are typed on the screen, A PF key may be used instead of the ENTER key.

Hints for opening multiple screen segments:

1. New screen segments can be started at any time and on any line of the screen with the ∇ command. This can be done either with or without passing text lines to the new screen segment.

2. If you enter a ∇ command where there are text lines showing below it, they will be processed as part of the new screen segment. This will normally add or replace text lines in the second function being edited. This is the recommended way to pass text lines from one function to another, or from an execute screen segment to another.

3. If you want to edit another function without passing it text lines, then blank out the screen segment below the ∇ command (with the ERASE EOF key) before entering it.

Hints for closing screen segments:

1. Screen segments can be closed with either the ∇ command or the [→] command. The closing ∇ can optionally be used to pass text lines to the adjacent screen segment. The action of the closing ∇ depends upon whether or not it is entered on a line by itself.

2. If the closing ∇ is used on a line by itself, then the screen segment will be erased, and no information will be passed to the adjacent segment.

3. If the closing ∇ is used at the end of a text line, then the screen segment will remain displayed, and be appended to an adjacent segment. This will normally add or replace text lines in the function being edited in the adjacent segment.

4. The entire editing session (all screen segments) can be aborted by two consecutive uses of the PA2 key.

Hints for using execute screen segments:

1. A screen segment may be opened with the name ⍎. this permits immediate execution of APL2 statements, instead of implicit line numbering. Only one statement at a time may be executed, and that is the one the cursor is on when the ENTER key or a PF key is used.

2. A function can be edited in one segment, and tested in a separate ⍎ segment on the screen at the same time. This requires use of the [∇] command in the normal segment to establish editing changes to the function in the workspace before trying them out.

3. Any lines displayed in an execute screen segment may be given bracketted line numbers. If the segment is then closed with a ∇ at the end of such a line, then the numbered

text lines showing on the screen will be appended to the adjacent screen segment.

The features of APL2 invite methods of problem solution that were previously less convenient to use. These and other defined functions and operators are provided in the supplied workspace *EXAMPLES*.

1. Trace the execution of a function:

```
     ∇ Z←L (F TRACE) R
[1]   ⍝  TRACE FUNCTION EXECUTION
[2]  →(0=⎕NC 'L')/V1
[3]   ⍝                ⍝  DYADIC CALL
[4]    (⊂L),⊂R         ⍝  DISPLAY BOTH ARGUMENTS
[5]    Z←L F R         ⍝  EXECUTE DYADIC FUNCTION
[6]    Z               ⍝  DISPLAY RESULT
[7]    →0
[8]   V1:              ⍝  MONADIC CALL
[9]    R               ⍝  DISPLAY RIGHT ARGUMENT
[10]   Z←F R           ⍝  EXECUTE MONADIC FUNCTION
[11]  Z                ⍝  DISPLAY RESULT
     ∇
```

This is a defined operator called *TRACE* which applies the function *F* to argument *R* if called monadically, or to arguments *L* and *R* if called dyadically. In either case, the argument(s) and the result are displayed as the function is applied.

Examples:

```
      Z ← 1 +TRACE 2
1 2
3
      Z
3

      Z ← +TRACE / 1 4 9
4 9
13
1 13
14
      Z
14
```

```
          Z ← +TRACE \ 1 4 9
1 4
5
4 9
13
1 13
14
          Z
1 5 14

          Z ← 2 +TRACE / 1 2 3 4
1 2
3
2 3
5
3 4
7
          Z
3 5 7
```

As the examples show, operators may be used to study functions as functions are used to study arrays.

2. Trap errors:

```
      ∇ Z←L (F TRAP) R
[1]    ⍝  TRAP AN ERROR
[2]   →(0=⎕NC 'L')/V1
[3]    ⍝  DYADIC CALL
[4]     Z←'⊂⎕EM' ⎕EA 'L F R'
[5]     →0
[6]    ⍝  MONADIC CALL
[7]    V1: Z←'⊂⎕EM' ⎕EA 'F R'
      ∇
```

This is a defined operator called *TRAP* which applies the function *F* to argument *R* if called monadically, or to arguments *L* and *R* if called dyadically. In either case, if application of the function causes an error, then the error message will be intercepted and returned (enclosed) in *Z*.

3. Force scalar conformability:

```
      ∇ Z←L (F PAD) R ;S
[1]    ⍝  CONFORM ALL AXES BY OVERTAKE
[2]     ⎕ES ((⍴⍴L)≠⍴⍴R)/5 2
[3]     S←(⍴L)⌈⍴R
[4]     Z←(S↑L) F S↑R
      ∇
```

This is a defined operator called *PAD* which applies dyadic function *F* to arguments *L* and *R* after padding them with fill elements till they are the same shape. Line 2 gives a

RANK ERROR if the ranks don't match. Line 3 computes the maximum length of each axis. Line 4 extends the arguments and applies the function.

Example:

```
      A ← 4 4ρ'WE  THEYUS  THEM'
      A
WE
THEY
US
THEM

      B ← 2 3ρ'WE OUR'
      B
WE
OUR

      A ∧.(=PAD) ⍉B
1 0
0 0
0 0
0 0
```

Note that the expression *A* ∧.= ⍉*B* would have given a *LENGTH ERROR*.

APPENDIX B. THE MIGRATION TRANSFER FORM

The migration transfer form is a simple character vector. It represents the name and value of a simple and non-mixed variable, or a displayable defined function. It is produced by the dyadic system function 1 ⎕*TF* *R*, where *R* is the name of the object.

The migration transfer form vector consists of four parts:

1. A data type code header character:

   '*F*' for a function

   '*N*' for a simple numeric array

   '*C*' for a simple character array

2. The name of the object, followed by a blank.

3. A character representation of the rank and shape of the array, followed by a blank.

4. A character representation of the array elements in row major order (any numeric conversions are done to 18 digits).

A defined function is treated as the character matrix of its canonical form, with semicolons between the local variable names in the header.

Examples:

```
      THIS ← 2 3⍴⍳6
      Z ← 1 ⎕TF 'THIS'
      Z
NTHIS 2 2 3 1 2 3 4 5 6
      ⍴Z
23

      THAT ← 3 4⍴'ABCDEFGHIJKL'
      Z ← 1 ⎕TF 'THAT'
      Z
CTHAT 2 3 4 ABCDEFGHIJKL
      ⍴Z
24
```

```
    ∇ Z←L PLUS R
[1]   Z←L+R
    ∇

      Z ← 1 □TF 'PLUS'
      Z
FPLUS 2 2 10 Z←L PLUS RZ←L+R
      ρZ
33

    ∇ V←PRIMES N;□IO M
[1]   □IO←1
[2]   M←⍳N
[3]   V←(1=0+.=(1↓M)∘.|M)/M
    ∇

      Z ← 1 □TF 'PRIMES'
      Z
FPRIMES 2 4 21 V←PRIMES N;□IO;M     □IO←1
        M←⍳N                     V←(1=0+.=(1↓M)∘.|M)/M
      ρZ
99
```

APPENDIX C. THE EXTENDED TRANSFER FORM

The extended transfer form is a simple character vector. It represents the name and value of a variable, or a displayable defined function or operator. It is produced by the monadic system function ⎕*TF R*, or the dyadic system function 2 ⎕*TF R*, where *R* is the name of the object.

If the object named by *R* is a variable which does not contain invalid characters, then its extended transfer form is a character vector which when executed generates the array. If *R* is a variable which does contain invalid characters, then its extended transfer is not executable, but it may be applied again to ⎕*TF*, which is its own inverse.

Examples:

```
      THIS ← 2 3ρι6
      Z ← 2 ⎕TF 'THIS'
      Z
THIS←2 3ρ1 2 3 4 5 6
      ρZ
20

      THAT ← 3 4ρ'ABCDEFGHIJKL'
      Z ← 2 ⎕TF 'THAT'
      Z
THAT←3 4ρ'ABCDEFGHIJKL'
      ρZ
23
```

If the object named by *R* is a variable which is a non-simple array, then its extended transfer form is a character vector which represents the array in a manner similar to vector notation. If there are multiple items in the array, then eack non-simple item is enclosed within parentheses.

Examples:

```
      THESE ← ⊂ι4
      Z ← 2 ⎕TF 'THESE'
      Z
THESE←⊂1 2 3 4
      ρZ
14

      THOSE ← 1 2ρ(⊂1 2 3),⊂2 2ρι4
      Z ← 2 ⎕TF 'THOSE'
      Z
THOSE←1 2ρ(1 2 3)(2 2ρ1 2 3 4)
      ρZ
30
```

```
      WE ← 'YOU' 'ME'
      Z ← 2 ⎕TF 'WE'
      Z
WE←'YOU' 'ME'
      ρZ
13
```

The extended transfer form of a shared variable or a system variable may be taken.

Example:

```
      Z ← 2 ⎕TF '⎕IO'
      Z
⎕IO←1
      ρZ
5
```

If the object named by *R* is a displayable defined function or operator with no set execution properties, then its extended transfer form is a character vector beginning with '⎕FX ', and followed by the representation for the vector-of-vectors variation of its canonical form.

Example:

```
    ∇ Z←L PLUS R
[1]   Z←L+R
    ∇

      Z ← 2 ⎕TF 'PLUS'
      Z
⎕FX 'Z←L PLUS R' 'Z←L+R'
      ρZ
24

    ∇ V←PRIMES N;⎕IO M
[1]   ⎕IO←1
[2]   M←⍳N
[3]   V←(1=0+.=(1↓M)∘.|M)/M
    ∇

      Z ← 2 ⎕TF 'PRIMES'
      Z
⎕FX 'V←PRIMES N;⎕IO M' '⎕IO←1' 'M←⍳N'
      'V←(1=0+.=(1↓M)∘.|M)/M'
      ρZ
61
```

If the object named by *R* is a displayable defined function or operator with any set execution properties, then its extended transfer form is a character vector beginning with the properties and '⎕FX ' and followed by the representation for the vector-of-vectors variation of its canonical form.

Example:

```
      0 1 1 0 ⎕FX 'Z←L PLUS R' 'Z←L+R'
PLUS

      Z ← 2 ⎕TF 'PLUS'
      Z
0 1 1 0 ⎕FX 'Z←L PLUS R' 'Z←L+R'
      ⍴Z
32
```

APPENDIX D. MIGRATION TO/FROM APL2

Migration to and from APL2 is accomplished with files containing the extended transfer forms of APL objects.

A transfer file has fixed length 80 character records. Each record has either ' ' or '*X*' in the first column. An APL object whose transfer form requires *N* characters is represented as ⌈*N*÷71 records in columns 2-72 of the transfer file, with blank (X'40') padding if necessary. All records but the last record begin with ' ' (blank). The last (or only) record of a transfer form in the file begins with '*X*'. Columns 73-80 of the transfer file are given sequence numbers on output by the system command)*OUT* or the *MIGRATE* workspace. The sequence numbers start with 00010000, and are incremented by 00010000. Columns 73-80 of the transfer file are ignored on input by the system command)*IN* or the *MIGRATE* workspace.

An example of a transfer file is shown in Figure 21 on page 324 as it would appear if displayed with ⎕*PW* equal to 50. The file has four records, and contains the operators *PAD* and *TRAP* that are described in "Appendix A. Further Examples" on page 313.

Either migration transfer forms or extended transfer forms of APL objects may be in the transfer file. Extended transfer forms for variables which appear in the file are preceded by '*A*'. Extended transfer forms for defined functions or operators which appear in the file are preceded by '*F*', the name of the function or operator, and a blank. The encoding of the characters in the files is as described in "The APL2 Character Set" on page 285.

```
 FPAD ⎕FX 'Z←L(F PAD)R;S' '⍝  CONFORM ALL AXES BY
X⍴R)/5 2' 'S←(⍴L)⌈⍴R' 'Z←(S↑L)F S↑R'
 FTRAP ⎕FX 'Z←L(F TRAP)R' '→(0=⎕NC ''L'')/V1' '⍝
 DIC CALL' 'Z←''⊂⎕EM'' ⎕EA ''L F R''' '→0' '⍝  MON
XM'' ⎕EA ''F R'''

      OVERTAKE' '⎕ES((⍴⍴L)≠⍴00010000
                            00020000
      TRAP AN ERROR' '⍝  DYA00030000
      ADIC CALL' 'V1:Z←''⊂⎕E00040000
                            00050000
```

Figure 21. A Transfer File

APPENDIX E. SUPPLIED WORKSPACES

Several utility workspaces are available with APL2.

MIGRATE This is an APL workspace which aids in the migration of workspaces to and from APL2 via migration files. The functions *INX* and *OUTX* perform operations similar to the system commands)*IN* and)*OUT* in APL2.

Functions, numeric arrays, and character arrays (containing any of the 256 characters in ⎕*AV*) may be migrated to APL2. Groups can not be migrated to APL2 with *MIGRATE*.

Functions (not containing special, national, or terminal control characters), simple numeric arrays, and simple character arrays (containing any of the 256 characters in ⎕*AV*) may be migrated from APL2. Operators, mixed arrays, and non-simple arrays can not be migrated from APL2 with *MIGRATE*.

The *DESCRIBE* variable in the workspace contains more documentation.

DISPLAY The *DISPLAY* function in this workspace produces a pictorial display of any array. *DISPLAY* uses the following graphics characters:

X'1B'	⎕*AV*[28]	┐	upper right corner
X'1C'	⎕*AV*[29]	┌	upper left corner
X'1E'	⎕*AV*[31]	└	lower left corner
X'1F'	⎕*AV*[32]	┘	lower right corner
X'2D'	⎕*AV*[46]	─	horizontal line
X'4F'	⎕*AV*[80]	│	vertical line

These characters may display differently on some terminals. Therefore, the *DISPLAYT* function creates the same pictures, except that the graphics characters used are for non-display terminals.

The result of either function is always a character matrix. The argument array and each item of it (except items in a simple array) is displayed in a box showing its rank, shape, emptiness, data type, and nesting.

The top and left box borders indicate rank. Either of the characters → or ↑ indicates that the dimension is at least one. Either of the characters ⊖ or ⌽ indicates that the dimension is zero. The absence of any of the characters → ↑ ⊖ or ⌽ in a top or left box border indicates that the dimension does not exist (that it is a scalar or vector).

The bottom box borders indicate data type. The character ~ indicates numeric, + indicates mixed, and ϵ indicates nested. The absence of any of the characters ~ + or ϵ in a bottom box border indicates that the data type is character.

Examples:

```
      A ← 1 'MORE' (2 'X')
      A ← A , (0 2⍴0) (2 2⍴⍳4) (5 '...')

      DISPLAY 2 3⍴A
┌→─────────────────────┐
↓┌─┐  ┌→───┐┌→──┐     │
││1│  │MORE││2 X│     │
│└~┘  └────┘└+──┘     │
│┌→──┐┌→──┐ ┌→───────┐│
│⌽0 0│↓1 2│ │┌─┐┌→──┐││
│└~──┘│3 4│ ││5││...│││
│     └~──┘ │└~┘└───┘││
│           └ϵ───────┘│
└ϵ────────────────────┘
```

```
      DISPLAYT 2 3⍴A
.→--------------------.
↓.-.  .→---..→--.     |
||1|  |MORE||2 X|     |
|'~'  '----''+--'     |
|.→--..→--. .→-------.|
|⌽0 0|↓1 2| |.-..→--.||
|'~--'|3 4| ||5||...|||
|     '~--' |'~''---'||
|           'ϵ-------'|
'ϵ--------------------'
```

EXAMPLES

This workspace contains a number of miscellaneous defined functions and operators of general use, including the ones described in "Appendix A. Further Examples" on page 313.

The comments in the individual functions and operators provide documentation. The functions can all be listed with the function *DUMP*, and the examples can all be run with the function *EXAMPLES*.

APPENDIX F. SYSTEM LIMITATIONS

These are limitations imposed on APL2 by the nature of its implementation:

1. The largest representable number in an array is 7.2370055773322621E75.

2. The smallest representable number in an array is ¯7.2370055773322621E75.

3. The most infinitesimal (near zero) representable numbers in an array are 5.397605346934027891E¯79 and ¯5.397605346934027891E¯79.

4. The maximum rank of an array is 64.

5. The maximum length of any dimension in an array is 2,147,483,647.

6. The maximum size of any simple array or simple item in a nested array is 16,777,216 bytes of storage.

7. The maximum number of elements in an array is 67,108,863.

8. The maximum depth of an array applied to any of the primitive functions Type (∊), Depth (≡), or Match (≡) is 98.

9. The maximum depth of a shared variable is 38.

10. The maximum depth of a copied variable is 38.

11. The maximum number of characters in the name of a shared variable is 255.

12. The maximum number of characters in a comment (less leading blanks) is 32,764.

13. The maximum number of lines in a defined function or operator is 2,147,483,647.

14. The maximum number of labels in a defined function or operator is 32,767.

15. The maximum number of local names (excluding labels) in a defined function or operator is 32,767.

16. The maximum number of slots in the internal symbol table is 32,767.

There are several features or operations in APL2 that can produce different results from those in APL. This list does not include extensions (operations which produced errors in APL but do not produce errors in APL2).

1. The Atomic Vector (⎕*AV*) is different. In particular, the alphabets are not contiguous.

2. ¯ and _ are alphanumeric characters.

3. ⎕*NC* name class 4 means operator.

4. ⎕*NC* name class ¯1 means invalid name.

5. The system function Name Class (⎕*NC*) applies to distinguished names (system variables and system functions).

6. The result of ⎕*EX*, ⎕*NC*, ⎕*SVO*, or ⎕*SVR* applied to a vector is a scalar.

7. The backspace character, the new line character (carriage return), and the line feed character are not permitted in a character constant or in function definition.

8. Lower case letters, new APL2 characters ⌼ ⍷ ⍸ ⌷ ⍂ ≡ ∵, national use characters ¢ | ! $ ¬ ¦ ` # @ " ~ { } \, and special characters & and % are permitted in character constants and comments.

9. The result of the system function Canonical Representation (⎕*CR*) separates local names in the function header with blanks.

10. The result of ⎕*CR* contains no unnecessary blanks in non-blank lines.

11. The result of ⎕*CR* may contain entirely blank lines.

12. Numeric constants in the canonical representation of a function show with the same precision with which they were entered.

13. The system function Fix (⎕*FX*) will accept blanks as the separator between local names in a function header.

14. Suspended or pendent defined functions may be expunged (with ⎕*EX*) or fixed (with ⎕*FX*).

15. Referencing ⍞ always produces a vector.

16. Referencing ⍞ after setting ⍞ with a prompt returns the composite of the prompt and the keyboard entry.

17. ⎕CT is an implicit argument of the function Residue (|).

18. ⎕CT is an implicit argument of the function Encode (⊤).

19. Negative integers are not in the domain of the dyadic Binomial (!) function.

20. An odd root of a negative number (like ¯8*÷3) is a complex number.

21. The result of ¯4○R is the negative square root if the argument R is negative.

22. The monadic format (⍕) or default display of a (simple) numeric matrix does not contain a leading column of blanks.

23. The monadic format (⍕) or default display of a (simple) numeric matrix has its columns formatted independently.

24. The result of dyadic format L⍕R, where L is a single non-zero integer, and R is less than 1, does not leave a blank for the units digit.

25. The display of a multi-dimensional array is folded at ⎕PW, if necessary, plane by plane, rather than line by line.

26. The display of an empty array having rank greater than 1 may use zero lines, or may extend to multiple lines.

27. The execution of the dyadic system function ⎕SVO is not necessarily atomic. If multiple shares are offered simultaneously, it is possible to exhaust the shared variable quota before all shares are fulfilled. In such a case, after a *SYSTEM LIMIT* error, some shares may be fulfilled while others are not.

28. If the left argument of the dyadic system function ⎕SVO is a 1-element vector, then it does not extend.

29. The dyadic system function Shared Variable Query (⎕SVQ) is not supported.

30. The edit command [⎕L] will display only line L of the function being edited. The command [⎕L-] will display from line L to the end of the function.

31. Changing the name of a function with the system editor creates a new function without affecting the original function.

32. Settings of Stop Control (*S*∆) and Trace Control (*T*∆) are not relocated as a result of line insertion or deletion by the system editor.

33. Statements entered in immediate execution which are interrupted by an error are placed in the *SI* stack, and may be resumed by entering →⍳0.

34. Groups are replaced by Indirect Copy and Indirect Erase.

APPENDIX H. APL2 UNDER CP/CMS

APL2 runs under CP/CMS by calling a module. Auxiliary processors may be named as arguments to the module. Upon both initiation and termination of an APL2 session, the module calls an EXEC named AP2EXIT. This EXEC may be modified to suit individual needs.

In APL2 under CP/CMS,

1. The method of signalling an interrupt varies with the type of terminal. The methods are shown in Figure 22 on page 334. A strong interrupt is signalled by two weak interrupts in succession.

2. The name of a workspace may have no more than eight characters.

3. An unspecified library number for the system commands *)LIB*, *)LOAD*, *)SAVE*, *)COPY*, and *)PCOPY* normally indicates a private library. Workspaces in a private library have a file type of APLWSV2.

4. The default (unspecified) library number is 1001. It can be defined to be a private, project, or public library number with an appropriate option in the APL2 command. This is also the value of the first element of the system variable Account Information (⎕*AI*[1]).

5. The file type of a public library workspace is Vnnnnnnn, where nnnnnnn is a 7-digit library number right adjusted and preceded with zeros. The maximum library number of a public library is 9999999.

6. A transfer file is specified by the file name, file type, and file mode separated by dots. The default file type is APLTF. The default file mode is A.

7. The value of the system variable Terminal Type (⎕*TT*) is 0.

8. Settings of the system variable Horizontal Tabs (⎕*HT*) are ignored, and the system resets ⎕*HT* to ⍳0 if it is set.

9. The maximum number of simultaneously shared variables, as reported by the *)QUOTA* system command, is 8+8×*APS*, where *APS* is the number of auxiliary processors active.

10. The size of shared memory, as reported by the *)QUOTA* system command, is about 500 bytes more than the maximum size of a shared variable.

11. The system commands *)OFF*, *)OFF HOLD*, *)CONTINUE*, and *)CONTINUE HOLD* normally return from APL2 to CMS, and do not

Terminal	Stop Output	Weak Interrupt
Typewriter	Attention	Attention
Display	PA2	ENTER or PA1 twice
Display with Session Manager	SUPPRESS	PA2

Figure 22. Terminal Attentions Under CP/CMS

perform a LOGOFF from CP/CMS. This behavior can be modified in the AP2EXIT EXEC.

12. The *MIGRATE* workspace is a VS APL workspace.

For more information, refer to the manual APL2 For CMS: Terminal User's Guide.

APPENDIX I. NATIONAL LANGUAGE TRANSLATIONS

System commands will be accepted, and system messages will be reported in any of several national languages according to the system variable National Language Translation (⎕*NLT*). System commands will always be accepted in English.

Tables of the various translations follow alphabetically as the language names appear in English. The system commands in each table are presented alphabetically as they appear in English, and the common system messages are presented alphabetically as they appear in the alternate national language.

DANISH LANGUAGE TRANSLATION

Danish System Commands

⎕NLT←'ENGLISH'	⎕NLT←'DANSK'
)CLEAR	)TOMT
)CONTINUE [HOLD]	)FORT [HOLD]
)COPY	)KOPI
)DROP	)FJERN
)EDITOR	)EDITOR
)ERASE	)SLET
)FNS	)FNR
)IN	)IND
)LIB	)BIB
)LOAD	)HENT
)MSG	)SEND
)MSGN	)SENDN
)NMS	)NVN
)OFF [HOLD]	)SLUT [HOLD]
)OPR	)OPR
)OPRN	)OPRN
)OPS	)OPTR
)OUT	)UD
)PBS	)SOS
)PCOPY	)BKOPI
)QUOTA	)KVOTA
)RESET	)RENS
)SAVE	)GEM
)SI	)SI
)SINL	)SINL
)SIS	)SIS
)SYMBOLS	)SYMBOLER
)VARS	)VAR
)WSID	)AAID

Danish System Messages

⎕NLT←'ENGLISH'	⎕NLT←'DANSK'
THIS WS IS CLEAR WS	AA IKKE NAVGIVET
INTERRUPT	AFBRYDELSE
WS NOT FOUND	ARBEJDSAREAL FINDES IKKE
WS FULL	ARBEJDSAREAL FYLDT
WS LOCKED	ARBEJDSAREAL L$ST
LIBRARY IN USE, RETRY	BIBLIOTEK I BRUG, PRəV IGEN
LIBRARY FULL	BIBLIOTEKS KVOTA BRUGT OP
DEFN ERROR	DEFINITIONSFEJL
THIS WS IS	DETTE AA HEDDER
IS	ER
IMPROPER LIBRARY REFERENCE	FORKERT BIBLIOTEKSKALD
AXIS ERROR	FORKERT AKSE
DOMAIN ERROR	FORKERT DOM*NE
INDEX ERROR	FORKERT INDEKS
INCORRECT COMMAND	FORKERT KOMMANDO
LENGTH ERROR	FORKERT L*NGDE
RANK ERROR	FORKERT RANG
VALENCE ERROR	FORKERT VALENS
SAVED	GEMT
GMT	GMT
NOT AN APL2 WS	IKKE ER APL2 ARBEJDSAREAL
NOT FOUND	IKKE FUNDET
NOT SAVED,	IKKE GEMT,
NOT COPIED	IKKE KOPIERET
NOT ERASED	IKKE SLETTET
ENTRY ERROR	INDL*SNINGSFEJL
VALUE ERROR	INGEN V*RDI
LIBRARY I/O ERROR	L*SE/SKRIVE-FEJL VED BIBLIOTEK
SI WARNING	SI əDELAGT
SYNTAX ERROR	SYNTAKSFEJL
SYSTEM ERROR	SYSTEM FEJL
SYSTEM LIMIT	SYSTEM GR*NSE
CLEAR WS	TOMT ARBEJDSAREAL
WAS	VAR
⎕__ ERROR	⎕__ FORKERT

FINNISH LANGUAGE TRANSLATION

Finnish System Commands

⎕NLT←'ENGLISH'	⎕NLT←'SUOMI'
)CLEAR	)TYHJENNYS
)CONTINUE [HOLD]	)JATKUU [PID#]
)COPY	)KOPIOI
)DROP	)TUHOA
)EDITOR	)EDITOR
)ERASE	)POISTA
)FNS	)FUNKTIOT
)IN	)TUO
)LIB	)KIRJASTO
)LOAD	)LATAA
)MSG	)SANOMA
)MSGN	)KERRO
)NMS	)NIMET
)OFF [HOLD]	)LOPETA [PID#]
)OPR	)OPSANOMA
)OPRN	)OPKERRO
)OPS	)OPER
)OUT	)VIE
)PBS	)APK
)PCOPY	)SKOPIOI
)QUOTA	)KIINTIəT
)RESET	)POIS
)SAVE	)TALLETA
)SI	)TI
)SINL	)TINL
)SIS	)TIE
)SYMBOLS	)SYMBOLIT
)VARS	)MUUTTUJAT
)WSID	)TTNIMI

Finnish System Messages

⎕NLT←'ENGLISH'	⎕NLT←'SUOMI'
RANK ERROR	ASTEVIRHE
NOT COPIED	EI KOPIOITU
NOT FOUND	EI LəYTYNYT
NOT AN APL2 WS	EI OLE APL2-TYəTILA
NOT ERASED	EI POISTETTU
NOT SAVED,	EI TALLETETTU,
ENTRY ERROR	EP*KELPO MERKKI SYəTəSS*
GMT	GMT
INDEX ERROR	INDEKSIVIRHE
SYSTEM LIMIT	J*RJESTELM*N RAJOITUKSEN YLITYS
SYSTEM ERROR	J*RJESTELM*N TOIMINTAH*IRIə
INTERRUPT	KESKEYTYS
SYNTAX ERROR	KIELIOPPIVIRHE
LIBRARY FULL	KIRJASTO ON T*YSI
LIBRARY IN USE, RETRY	KIRJASTO ON VARATTUNA, YRIT* UUDELLEEN
DEFN ERROR	OHJELMAN M**RITYSVIRHE
WAS	OLI
IS	ON
LENGTH ERROR	PITUUSVIRHE
DOMAIN ERROR	SOPIMATON ARGUMENTTI
AXIS ERROR	SUUNTAVIRHE
SAVED	TALLETETTU
SI WARNING	TI-VAROITUS
LIBRARY I/O ERROR	TIEDONSIIRTOVIRHE
VALUE ERROR	TUNTEMATON NIMI TAI PUUTTUVA ARVO
WS LOCKED	TYəTILA ON LUKITTU
WS FULL	TYəTILA T*YNN*
WS NOT FOUND	TYəTILAA EI LəYTYNYT
THIS WS IS CLEAR WS	TYəTILALLA EI OLE NIME*
THIS WS IS	TYəTILAN NIMI ON
CLEAR WS	TYHJ* TY TILA
IMPROPER LIBRARY REFERENCE	VIRHEELLINEN KIRJASTOVIITE
INCORRECT COMMAND	VIRHEELLINEN KOMENTO
VALENCE ERROR	VIRHEELLINEN M**R* ARGUMENTTEJA
⎕__ ERROR	⎕__ VIRHE

FRENCH LANGUAGE TRANSLATION

French System Commands

⎕NLT←'ENGLISH'	⎕NLT←'FRANCAIS'
)CLEAR	)LIBER
)CONTINUE [HOLD]	)SUSP [APL]
)COPY	)COPIER
)DROP	)ELIM
)EDITOR	)EDITEUR
)ERASE	)EFFACER
)FNS	)FNS
)IN	)LECT
)LIB	)BIB
)LOAD	)CHARGER
)MSG	)MSG
)MSGN	)MSGN
)NMS	)NOMS
)OFF [HOLD]	)FIN [APL]
)OPR	)OPER
)OPRN	)OPERN
)OPS	)OPERS
)OUT	)ECRIT
)PBS	)EAI
)PCOPY	)PCOPIER
)QUOTA	)QUOTA
)RESET	)RESTAUR
)SAVE	)SAUV
)SI	)IE
)SINL	)IELN
)SIS	)CIE
)SYMBOLS	)SYMB
)VARS	)VARS
)WSID	)ZONE

French System Messages

⎕NLT←'ENGLISH'	⎕NLT←'FRANCAIS'
LIBRARY IN USE, RETRY	BIBLIOTHEQUE UTILISEE, ESSAYEZ DE NOUVEAU
SYSTEM LIMIT	CAPACITE DU SYSTEME DEPASSEE
ENTRY ERROR	CARACTERE NON VALIDE
THIS WS IS CLEAR WS	CETTE ZONE EST LIBRE
THIS WS IS	CETTE ZONE S'APPELLE
INCORRECT COMMAND	COMMANDE INCORRECTE
DEFN ERROR	ERREUR DE DEFINITION
LENGTH ERROR	ERREUR DE DIMENSION
DOMAIN ERROR	ERREUR DE DOMAINE
RANK ERROR	ERREUR DE RANG
SYNTAX ERROR	ERREUR DE SYNTAXE
VALENCE ERROR	ERREUR DE VALENCE
SYSTEM ERROR	ERREUR DU SYSTEME
AXIS ERROR	ERREUR D'AXE
INDEX ERROR	ERREUR D'INDEXATION
LIBRARY I/O ERROR	ERREUR E-S EN BIBLIOTHEQUE
IS	EST
WAS	ETAIT
GMT	GMT
SI WARNING	INDICATEUR D'ETAT ENDOMMAGE
INTERRUPT	INTERRUPTION
LIBRARY FULL	NOMBRE DE ZONES DEPASSE
NOT ERASED	NON EFFACE
NOT SAVED,	NON SAUVEGARDE
NOT COPIED	OBJETS NON COPIES
NOT FOUND	OBJETS NON TROUVES
IMPROPER LIBRARY REFERENCE	REFERENCE INCORRECTE A LA BIBLIOTHEQUE
SAVED	SAUVEGARDE
VALUE ERROR	VALEUR NON DEFINIE
WS FULL	ZONE DE TRAVAIL PLEINE
CLEAR WS	ZONE LIBRE
NOT AN APL2 WS	ZONE NON MISE EN FORME POUR APL2
WS NOT FOUND	ZONE NON TROUVEE
WS LOCKED	ZONE PROTEGEE
⎕__ ERROR	⎕__ EN ERREUR

GERMAN LANGUAGE TRANSLATION

German System Commands

⎕NLT←'ENGLISH'	⎕NLT←'DEUTSCH'
)CLEAR	)LEERE
)CONTINUE [HOLD]	)WEITER [HALTE]
)COPY	)KOPIERE
)DROP	)ENTFERNE
)EDITOR	)EDITOR
)ERASE	)LOESCHE
)FNS	)FUN
)IN	)EIN
)LIB	)BIBL
)LOAD	)LADE
)MSG	)ANFRAGE
)MSGN	)NACHRICHT
)NMS	)NAM
)OFF [HOLD]	)ENDE [HALTE]
)OPR	)OPRANFR
)OPRN	)OPRNACHR
)OPS	)OPE
)OUT	)AUS
)PBS	)RSZ
)PCOPY	)SKOPIERE
)QUOTA	)QUOTEN
)RESET	)GRUNDSTELLUNG
)SAVE	)SPEICHERE
)SI	)SI
)SINL	)SINL
)SIS	)SIA
)SYMBOLS	)SYMBOLE
)VARS	)VAR
)WSID	)ABNAME

German System Messages

⎕NLT←'ENGLISH'	⎕NLT←'DEUTSCH'
WS LOCKED	AB GESPERRT
THIS WS IS CLEAR WS	AB HAT KEINEN NAMEN
THIS WS IS	AB NAME IST
WS NOT FOUND	AB NICHT GEFUNDEN
CLEAR WS	AB OHNE NAME
LIBRARY FULL	AB QUOTE AUSGESCHOEPFT
WS FULL	AB VOLL
LIBRARY IN USE, RETRY	BIBLIOTHEK BENUTZT, WIEDERHOLE
DEFN ERROR	DEFINITIONSFEHLER
LIBRARY I/O ERROR	E/A FEHLER BEI BILIOTHEKSZUGRIFF
VALENCE ERROR	FALSCHE ARGUMENTANZAHL
SAVED	GESPEICHERT
INDEX ERROR	INDEXFEHLER
IS	IST
NOT AN APL2 WS	KEIN APL2 AB
AXIS ERROR	KOORDINATENFEHLER
LENGTH ERROR	LAENGENFEHLER
VALUE ERROR	NAME OHNE WERT
NOT FOUND	NICHT GEFUNDEN
NOT ERASED	NICHT GELOESCHT
NOT SAVED,	NICHT GESPEICHERT,
NOT COPIED	NICHT KOPIERT
RANK ERROR	RANGFEHLER
SI WARNING	SI WARNUNG
SYNTAX ERROR	SYNTAXFEHLER
SYSTEM LIMIT	SYSTEMBESCHRAUENKUNG
SYSTEM ERROR	SYSTEMFEHLER
IMPROPER LIBRARY REFERENCE	UNERLAUBTER BIBLIOTHEKSZUGRIFF
INCORRECT COMMAND	UNGUELTIGE SYSTEMANWEISUNG
DOMAIN ERROR	UNGUELTIGES ARGUMENT
ENTRY ERROR	UNGUELTIGES ZEICHEN
INTERRUPT	UNTERBRECHUNG
WAS	WAR
GMT	WEZ
⎕__ ERROR	⎕__ FEHLER

NORWEGIAN LANGUAGE TRANSLATION

Norwegian System Commands

⎕NLT←'ENGLISH'	⎕NLT←'NORSK'
)CLEAR	)NULLSTILL
)CONTINUE [HOLD]	)FORTSETT [HOLD]
)COPY	)KOPIER
)DROP	)FJERN
)EDITOR	)EDITOR
)ERASE	)SLETT
)FNS	)FUNKSJONER
)IN	)INN
)LIB	)BIBLIOTEK
)LOAD	)HENT
)MSG	)MELDING
)MSGN	)MELDINGN
)NMS	)NMS
)OFF [HOLD]	)AV [HOLD]
)OPR	)OPR
)OPRN	)OPRN
)OPS	)OPS
)OUT	)UT
)PBS	)PBS
)PCOPY	)BKOPIER
)QUOTA	)KVOTE
)RESET	)RESET
)SAVE	)LAGRE
)SI	)SI
)SINL	)SINL
)SIS	)SIS
)SYMBOLS	)SYMBOLER
)VARS	)VARIABLER
)WSID	)AOID

Norwegian System Messages

⎕NLT←'ENGLISH'	⎕NLT←'NORSK'
AXIS ERROR	AKSEFEIL
WS NOT FOUND	AO IKKE FUNNET
WS LOCKED	AO L$ST
THIS WS IS CLEAR WS	AO NULLSTILT
CLEAR WS	AO NULLSTILT
LIBRARY FULL	AO-KVOTE OPPBRUKT
WS FULL	ARBEIDSOMR$DE FULLT
INTERRUPT	AVBRUDD
LIBRARY I/O ERROR	BIBLIOTEK I/O FEIL
LIBRARY IN USE, RETRY	BIBLIOTEK OPPTATT, PR∂V IGJEN
DEFN ERROR	DEFINISJONSFEIL
THIS WS IS	DETTE ER
IS	ER
INCORRECT COMMAND	FEILAKTIG KOMMANDO
GMT	GMT
NOT AN APL2 WS	IKKE ET APL2 AO
NOT FOUND	IKKE FUNNET
NOT COPIED	IKKE KOPIERT
NOT SAVED,	IKKE LAGRET,
NOT ERASED	IKKE SLETTET
INDEX ERROR	INDEKSFEIL
SAVED	LAGRET
LENGTH ERROR	LENGDE-KONFLIKT
DOMAIN ERROR	OMR$DEFEIL
RANK ERROR	RANG-KONFLIKT
SI WARNING	SI ∂DELAGT
SYNTAX ERROR	SYNTAKSFEIL
SYSTEM LIMIT	SYSTEM GRENSE
SYSTEM ERROR	SYSTEMFEIL
ENTRY ERROR	TASTFEIL
IMPROPER LIBRARY REFERENCE	UGYLDIG BIBLIOTEK-REFERANSE
VALENCE ERROR	UGYLDIG FUNKSJON
WAS	VAR
VALUE ERROR	VERDIFEIL
⎕__ ERROR	⎕__ FEIL

SPANISH LANGUAGE TRANSLATION

Spanish System Commands

⎕NLT←'ENGLISH'	⎕NLT←'ESPANOL'
)CLEAR	)LIMPIAR
)CONTINUE [HOLD]	)CONTINUAR [MANTENER]
)COPY	)COPIAR
)DROP	)ELIM
)EDITOR	)EDITOR
)ERASE	)BORRAR
)FNS	)FNS
)IN	)TRAER
)LIB	)BIB
)LOAD	)CARGAR
)MSG	)MSJ
)MSGN	)MSJN
)NMS	)NMS
)OFF [HOLD]	)DESCONECTAR [MANTENER]
)OPR	)OPR
)OPRN	)OPRN
)OPS	)OPS
)OUT	)LLEVAR
)PBS	)REI
)PCOPY	)COPIARP
)QUOTA	)QUOTA
)RESET	)LIBERAR
)SAVE	)ARCHI
)SI	)IP
)SINL	)IPV
)SIS	)IPS
)SYMBOLS	)SIMBOLOS
)VARS	)VARS
)WSID	)NOMBRE

Spanish System Messages

⎕NLT←'ENGLISH'	⎕NLT←'ESPANOL'
SAVED	ARCHIVADO
LIBRARY IN USE, RETRY	BIBLIOTECA UTILIZANDOSE, REPITA
LIBRARY FULL	CUOTA DE ET EXCEDIDA
WAS	ERA
DEFN ERROR	ERROR DE DEFINICION
DOMAIN ERROR	ERROR DE DOMINIO
AXIS ERROR	ERROR DE EJES
ENTRY ERROR	ERROR DE ESCRITURA
INDEX ERROR	ERROR DE INDICE
LENGTH ERROR	ERROR DE LONGITUD
RANK ERROR	ERROR DE RANGO
SYNTAX ERROR	ERROR DE SINTAXIS
VALENCE ERROR	ERROR DE VALENCIA
VALUE ERROR	ERROR DE VALOR
⎕__ ERROR	ERROR DE ⎕__
SYSTEM ERROR	ERROR DEL SISTEMA
LIBRARY I/O ERROR	ERROR E/S DE BILIOTECA
IS	ES
WS FULL	ESPACIO DE TRABAJO LLENO
NOT AN APL2 WS	ESPACIO DE TRABAJO NO APL2
THIS WS IS	ESTE ET ES
THIS WS IS CLEAR WS	ESTE ET ES ANONIMO
CLEAR WS	ET ANONIMO
WS LOCKED	ET NECESITA PALABRA CLAVE
WS NOT FOUND	ET NO HALLADO
GMT	GMT
INTERRUPT	INTERRUPCION
SI WARNING	IP ALTERADA
SYSTEM LIMIT	LIMITE DEL ERROR
INCORRECT COMMAND	MANDATO INCORRECTO
NOT SAVED,	NO ARCHIVADO,
NOT ERASED	NO BORRADO
NOT COPIED	NO COPIADO
NOT FOUND	NO ENCONTRADO
IMPROPER LIBRARY REFERENCE	NUMERO DE BILIOTECA INCORRECTO

SWEDISH LANGUAGE TRANSLATION

Swedish System Commands

⎕NLT←'ENGLISH'	⎕NLT←'SVENSKA'
)CLEAR	)NY
)CONTINUE [HOLD]	)FORTS [H$LL]
)COPY	)KOPIERA
)DROP	)KASTA
)EDITOR	)REDIGERA
)ERASE	)RADERA
)FNS	)FUNK
)IN	)IN
)LIB	)BIBL
)LOAD	)LADDA
)MSG	)MEDD
)MSGN	)MEDDN
)NMS	)NAMN
)OFF [HOLD]	)SLUT [H$LL]
)OPR	)OPR
)OPRN	)OPRN
)OPS	)OPER
)OUT	)UT
)PBS	)PBS
)PCOPY	)SKOPIERA
)QUOTA	)KVOT
)RESET	)RENSA
)SAVE	)SPARA
)SI	)SI
)SINL	)SINL
)SIS	)SIS
)SYMBOLS	)SYMB
)VARS	)VAR
)WSID	)ID

Swedish System Messages

⎕NLT←'ENGLISH'	⎕NLT←'SVENSKA'
WS NOT FOUND	ARBETSAREAN EJ FUNNEN
WS FULL	ARBETSAREAN FULL
THIS WS IS	ARBETSAREAN HETER
WS LOCKED	ARBETSAREAN L$ST
THIS WS IS CLEAR WS	ARBETSAREAN SAKNAR NAMN
INTERRUPT	AVBRROTT
AXIS ERROR	AXIS-FEL
LIBRARY FULL	BIBLIOTEKET FULLT
LIBRARY IN USE, RETRY	BIBLIOTEKET UPPTAGET, FəRSəK IGN
DEFN ERROR	DEFINITIONS-FEL
NOT AN APL2 WS	EJ EN APL2 ARBETSAREA
NOT FOUND	EJ FUNNA
NOT COPIED	EJ KOPIERADE
NOT ERASED	EJ RADERADE
NOT SAVED,	EJ SPARAD,
⎕__ ERROR	FEL I ⎕__
LIBRARY I/O ERROR	FEL VID BIBLIOTEKS-I/O
INCORRECT COMMAND	FELAKTIGT KOMMANDO
GMT	GMT
INDEX ERROR	INDEX-FEL
LENGTH ERROR	L*NGD-FEL
ENTRY ERROR	L*S-FEL
IMPROPER LIBRARY REFERENCE	OGILTIG BIBLIOTEKS-REFERENS
DOMAIN ERROR	OMR$DES-FEL
CLEAR WS	NY ARBETSAREA
RANK ERROR	RANG-FEL
SI WARNING	SI FəRSTəRT
SAVED	SPARAD
SYNTAX ERROR	SYNTAX-FEL
SYSTEM LIMIT	SYSTEM-BEGR*ANSNING
SYSTEM ERROR	SYSTEM-FEL
WAS	TIDIGARE
VALENCE ERROR	VALENS-FEL
VALUE ERROR	V*RDE-FEL
IS	*R

ALPHABETIC CHARACTER A character which is a capital letter, an underscored capital letter, or ∆ or ⍙.

ALPHANUMERIC CHARACTER A character which is alphabetic, a digit, or ¯ or _.

AMBI-VALENT FUNCTION A function name which represents both a monadic and dyadic function. The one intended is determined from context.

ARGUMENT An array parameter that is passed to a function (to be distinguished from an operand of an operator).

ARRAY A rectangular collection of data elements. An array has rank (possibly 0), and shape (possibly empty).

BODY All lines after the first (line 0) of a defined function or operator.

CONSTANT A scalar or vector, either character or numeric, that appears explicitly in an APL2 statement. A constant always has the same value.

DERIVED FUNCTION A function which is the result of applying an operator to one or two operands in its domain.

DYADIC FUNCTION A function which is defined for both a left and a right argument.

DYADIC OPERATOR An operator which is defined for both a left operand and a right operand.

EBCDIC Extended Binary Coded Decimal Interchange Code.

ELEMENT A scalar which appears in an array. An element may be a character, a number, or an enclosed array.

EMPTY ARRAY An array which has a 0 in its shape.

EXPLICIT RESULT The array value returned from a primitive function, or the array value returned from a defined function or operator by having it assigned in both the header and the body.

EXPRESSION A sequence of one or more syntactic tokens, which may be symbols or names representing arrays (constants or variables), functions, and operators.

FILL ELEMENT The scalar which is used by the functions Expand, Replicate, and Take. It is either 0 or ' ', or a nested scalar array containing only 0 and ' '. It is determined by the expression ⊂∊⊃*R*.

FUNCTION An operation which takes one or two arrays as explicit arguments, and produces an array as a result.

FUZZ The tolerance used in computing an equality.

HALTED Suspended or pendent. Said of a defined function or operator.

HEADER The first line (line 0) of a defined function or operator.

INTEGER A (whole) number with no fractional or imaginary part.

ITEM A disclosed element of an array. An item may have any rank, and is the data within the scalar structure of an element.

LOGICAL ARRAY An array containing only 0, 1, or both.

MATRIX An array with rank equal to 2.

MIXED ARRAY A simple array which contains both characters and numbers.

MONADIC FUNCTION A function which is defined for only a right argument.

MONADIC OPERATOR An operator which is defined for only a left operand.

NESTED ARRAY A non-simple array.

NILADIC FUNCTION A function which is defined for no arguments. A niladic function may not be used as the function operand of an operator.

NON-SCALAR ARRAY An array with rank greater than 0.

NON-SIMPLE ARRAY A non-empty array in which at least one item is not a scalar character or number, or an empty array in which the prototype is a non-scalar array.

OPERAND A function or array parameter that is passed to an operator (to be distinguished from an argument of a function).

OPERATOR An operation which takes one or two functions or arrays as operands and produces a derived function.

PENDENT Halted, and remaining in an incomplete state, but not directly restartable. Said of a defined function or operator. A pendent function or operator has invoked another defined function or operator.

PERVASIVE FUNCTION A function which applies independently to all the simple scalars in its arguments, and produces a result of structure similar to that of its arguments. A pervasive function distributes over the function Pick (⊃).

PRECEDENCE Priority of importance.

PROTOTYPE The Type of the First of an array (∊⊃*A*). If the array is empty, then this is equivalent to ⊃*A*, and is its disclosed structure.

RANK The number of dimensions of an array.

REAL NUMBER A number with no imaginary part, or an imaginary part of 0.

SCALAR An array with rank equal to 0.

SCALAR FUNCTION A function which applies independently to all the scalars in its arguments, and produces a result of <u>shape</u> similar to that of its arguments. A scalar function distributes over Bracket Indexing.

SCOPE Range of influence.

SESSION VARIABLE A system variable which, if assigned a valid global value, will persist over a workspace clear or load. An invalid value assigned to a session variable is ignored.

SHAPE The collection of the lengths of all the dimensions of an array.

SIMPLE ARRAY A non-empty array in which all items are scalar characters or numbers, or an empty array in which the prototype is a simple scalar array.

STRUCTURE The arrangement and type of simple scalars in an array. In a simple array, the structure is the shape. In a non-simple array, the structure is the shape, together with the structure of each item in the array.

SUB-ARRAY An array which is a contiguous subset along one or more axes of an array. The subset may be proper or improper. That is, the sub-array may be of equal or smaller rank and shape compared with the array itself.

SUSPENDED Halted, and remaining in an incomplete state, and directly restartable. Said of a defined function or operator. The halt may have been caused by an error, an attention, or a stop control.

UNIFORM ARRAY An array in which all items have the same structure.

VALENCE The number of explicit arguments that a function takes, or the number of explicit operands that an operator takes. For example, a niladic function has a valence of 0, a monadic function has a valence of 1, and a dyadic function has a valence of 2.

VECTOR An array with rank equal to 1.

VARIABLE A named array.

WORKSPACE The common organizational unit of the APL2 system. It contains data, programs, and execution status.

Brown, J.A., "A Generalization of APL", Doctoral Thesis, 1971, Dept. of Computer and Information Science, Syracuse University, New York, Clearing House 74H004942 AD-770488.

Brown, J.A., "APL Language Extensions", Proceedings of SEAS 1978 anniversary meeting, Stresa, Italy, Vol. 1, pp. 335-353.

Brown, J.A., "Evaluating Extensions to APL", APL Quote Quad, Vol. 9, No.4-part 1, June, 1979, pp. 148-155.

Falkoff, A.D. and Iverson, K.E., "Communication in APL Systems", IBM Philadelphia Scientific Center report 320-3022, May, 1973.

Falkoff, A.D. and Iverson, K.E., "The Design of APL", IBM Journal of Research and Development, Vol. 17, no. 4, July, 1973.

Falkoff, A.D., "An APL Standard", APL Quote Quad, Vol. 9, No.4-part 2, June, 1979, pp. 415-453.

Ghandour, Z. and Mezei J., "General Arrays, Operators and Functions", IBM Journal of Research and Development, Vol. 17, no. 4, July, 1973.

Gull, W.E. and Jenkins, M.A., "Recursive Data Structures in APL", CACM vol. 22, no. 4, February, 1979 pp. 79-96.

Gull, W.E. and Jenkins, M.A., "Decisions for Type in APL", Proceedings of the 6th Symposium on the Principles of Programming Languages, San Antonio, Texas, January, 1979.

Haegi, H.R., "The Extension of APL to Treelike Data Structures", APL Quote Quad, Vol. 7, no. 2.

Iverson, K.E., "APL as an Analytic Notation", Proceedings of APL V, May 17, 1973, Toronto, Canada.

Iverson, K.E., "Operators and Functions", IBM Research report 7091, Yorktown Heights, New York.

Jenkins, M.A. and Gull, W.E., "Recursive Data Structures and Related Control Mechanisms in APL", Proceedings of APL76, Ottawa, Canada.

Lathwell, R.H. and Mezei, J.E., "A Formal Description of APL", IBM Philadelphia Scientific Center Report 320-3008, November, 1971.

Lathwell, R.H., "The APL Shared Variable System", IBM Journal of Research and Development, Vol. 17, No. 4, July, 1973.

Lathwell, R.H., "System Formulation and APL Shared Variables", IBM Journal of Research and Development, Vol. 17, No. 4, July, 1973.

McDonnell, E.E., "Integer Functions of Complex Numbers, with Applications", IBM Philadelphia Scientific Center report 320-3015, February, 1973.

Mezei, J.E., "Uses of General Arrays and Operators", Proceedings of 6th international APL users conference, Anaheim, California, May, 1974.

Mercer, R., "Extensions of APL to include Arrays of Arrays", University Computing Center Report, University of Massachusetts, Amherst.

More, T., "Notes on the Development of a Theory of Arrays", Philadelphia Scientific Center report 320-3016, May, 1973.

More, T., "Notes on the Axioms for a Theory of Arrays", Philadelphia Scientific Center report 320-3017, May, 1973.

More, T., "Axioms and Theorems for a Theory of Arrays", IBM Journal of Research and Development, Vol. 17, no. 2, March, 1973.

More, T., "A Theory of Arrays with Applications to Databases", IBM Cambridge Scientific Center report G320-2107, September, 1975.

More, T., "Types and Prototypes in a Theory of Arrays", IBM Cambridge Scientific Center report G320-2112, May, 1976.

More, T., "On the Composition of Array-theoretic Operations", IBM Cambridge Scientific Center report G320-2113, May, 1976.

More, T., "Notes on the Diagrams, Logic, and Operations of Array Theory", in "Structure and Operations in Engineering and Management Systems", Bjorke and Franksen, editors, Norwegian institute of Technology, Tapir Publishers, 1981.

Orth, D.L., "A User's View of General Arrays", IBM Watson Research Center report RC 8782, April 1981.

Orth, D.L., "Empty Arrays in Extended APL", IBM Watson Research Center, January, 1982.

Penfield, P., "Design Choices for Complex APL", APL Quote Quad, Vol. 8, no. 3, March, 1977.

Quine, W., "Mathematical Logic", revised edition, Harvard University Press, 1940.

Rabenhorst, D.A., "APL Function Variants and System Labels", IBM Watson Research Center, February, 1982.

Wilkinson J.H., "Linear Algebra", C. Reinsch Springer-Verlag, New York, 1971.

ACKNOWLEDGEMENTS

The following people have contributed to the content of the APL2 Language or this manual:

Doug Aiton, Ev Allen, Phil Benkard, Larry Breed, Norm Brenner, Jim Brown, Graham Driscoll, Dick Doyle, Dick Dunbar, Ed Eusebi, Adin Falkoff, John Gerth, Jean-Jacques Girardot, Alan Graham, Bruce Hartigan, Dick Harvey, Ken Iverson, Mike Jenkins, Dieter Lattermann, Gary Logan, Blair Martin, Marilyn Mattei, Gene McDonnell, Jon McGrew, Eben Moglen, Mike Montalbano, Trenchard More, Alex Morrow, Andre Orlans, Don Orth, Ray Polivka, Dave Rabenhorst, Stan Schmidt, Howard Smith, Karl Soop, Ray Trimble, Mike Van der Meulen, Mike Wheatley, Ron Wilks.

The following people have contributed heavily to the implementation of the APL2 Language:

Doug Aiton, Ev Allen, Jim Brown (especially), John Gerth, Eben Moglen, Dave Rabenhorst, Mike Wheatley.

Appreciation is given to those who wrote the original documentation of APL, upon which parts of this manual are based.

This manual was written by Dave Rabenhorst.

SB21-3015-0

APL 2 Language Manual Printed in U.S.A. SB21-3015-0

APL 2
Language Manual

SB21-3015-0

READER'S COMMENT FORM

You may use this form to communicate your comments about this publication, its organization, or subject matter, with the understanding that IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

Your comments will be sent to the author's department for whatever review and action, if any, are deemed appropriate.

Note: *Copies of IBM publications are not stocked at the location to which this form is addressed. Please direct any requests for copies of publications, or for assistance in using your IBM system, to your IBM representative or to the IBM branch office serving your locality.*

Possible topics for comment are:

Clarity Accuracy Completeness Organization Coding Retrieval Legibility

If you wish a reply, give your name, company, mailing address, and date:

____

____

____

____

What is your occupation? ______________________________

Number of latest Newsletter associated with this publication: ____________________

Thank you for your cooperation. No postage stamp necessary if mailed in the U.S.A. (Elsewhere, an IBM office or representative will be happy to forward your comments or you may mail directly to the address in the Edition Notice on the back of the title page.)

Note: Staples can cause problems with automated mail sorting equipment.
Please use pressure sensitive or other gummed tape to seal this form.

SB21-3015-0

Reader's Comment Form

Fold and tape

Please Do Not Staple

Fold and tape

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO. 40 ARMONK, N.Y.

POSTAGE WILL BE PAID BY ADDRESSEE:

International Business Machines Corporation
Department 824
1133 Westchester Avenue
White Plains, New York 10604

Fold and tape

Please Do Not Staple

Fold and tape

IBM®

Cut or Fold Along Line

APL 2
Language Manual

SB21-3015-0

READER'S COMMENT FORM

You may use this form to communicate your comments about this publication, its organization, or subject matter, with the understanding that IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

Your comments will be sent to the author's department for whatever review and action, if any, are deemed appropriate.

Note: *Copies of IBM publications are not stocked at the location to which this form is addressed. Please direct any requests for copies of publications, or for assistance in using your IBM system, to your IBM representative or to the IBM branch office serving your locality.*

Possible topics for comment are:

Clarity   Accuracy   Completeness   Organization   Coding   Retrieval   Legibility

If you wish a reply, give your name, company, mailing address, and date:

____

____

____

____

What is your occupation? ________________________________

Number of latest Newsletter associated with this publication: ________________

Thank you for your cooperation. No postage stamp necessary if mailed in the U.S.A. (Elsewhere, an IBM office or representative will be happy to forward your comments or you may mail directly to the address in the Edition Notice on the back of the title page.)

Note: Staples can cause problems with automated mail sorting equipment.
Please use pressure sensitive or other gummed tape to seal this form.

SB21-3015-0

Reader's Comment Form

Fold and tape   Please Do Not Staple   Fold and tape

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS   PERMIT NO. 40   ARMONK, N.Y.

POSTAGE WILL BE PAID BY ADDRESSEE:

International Business Machines Corporation
Department 824
1133 Westchester Avenue
White Plains, New York 10604

Fold and tape   Please Do Not Staple   Fold and tape

IBM®

Cut or Fold Along Line

APL 2
Language Manual

SB21-3015-0

READER'S COMMENT FORM

You may use this form to communicate your comments about this publication, its organization, or subject matter, with the understanding that IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

Your comments will be sent to the author's department for whatever review and action, if any, are deemed appropriate.

Note: *Copies of IBM publications are not stocked at the location to which this form is addressed. Please direct any requests for copies of publications, or for assistance in using your IBM system, to your IBM representative or to the IBM branch office serving your locality.*

Possible topics for comment are:

Clarity Accuracy Completeness Organization Coding Retrieval Legibility

If you wish a reply, give your name, company, mailing address, and date:

____

____

____

____

What is your occupation? ______________________________

Number of latest Newsletter associated with this publication: ____________________

Thank you for your cooperation. No postage stamp necessary if mailed in the U.S.A. (Elsewhere, an IBM office or representative will be happy to forward your comments or you may mail directly to the address in the Edition Notice on the back of the title page.)

Note: Staples can cause problems with automated mail sorting equipment.
Please use pressure sensitive or other gummed tape to seal this form.

SB21-3015-0

Reader's Comment Form

Cut or Fold Along Line

Fold and tape

Please Do Not Staple

Fold and tape

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 40 ARMONK, N.Y.

POSTAGE WILL BE PAID BY ADDRESSEE:

International Business Machines Corporation
Department 824
1133 Westchester Avenue
White Plains, New York 10604

Fold and tape

Please Do Not Staple

Fold and tape

IBM®

APL 2 Language Manual Printed in U.S.A. SB21-3015-0